inference.py

```python
from PIL import Image
import torch
import time
import psutil
import os

from text_image_token_processor_1 import PaliGemmaProcessor
from decoder_1 import KVCache, PaliGemmaForConditionalGeneration
from utils import load_hf_model



def move_inputs_to_device(model_inputs: dict, device: str):
    model_inputs = {k: v.to(device) for k, v in model_inputs.items()}
    return model_inputs



def get_model_inputs(
    processor: PaliGemmaProcessor, prompt: str, image_file_path: str, device: str
):
    image = Image.open(image_file_path)
    images = [image]
    prompts = [prompt]
    model_inputs = processor(text=prompts, images=images)
    model_inputs = move_inputs_to_device(model_inputs, device)
    return model_inputs



def test_inference(
    model: PaliGemmaForConditionalGeneration,
    processor: PaliGemmaProcessor,
    device: str,
    prompt: str,
    image_file_path: str,
    max_tokens_to_generate: int,
    temperature: float,
    top_p: float,
    do_sample: bool,
):
    model_inputs = get_model_inputs(processor, prompt, image_file_path, device)
    input_ids = model_inputs["input_ids"]
    attention_mask = model_inputs["attention_mask"]
    pixel_values = model_inputs["pixel_values"]

    kv_cache = KVCache()

    # Generate tokens until you see the stop token
    stop_token = processor.tokenizer.eos_token_id
    generated_tokens = []
    start_time = time.time()
    for _ in range(max_tokens_to_generate):
        outputs = model(
            input_ids=input_ids,
            pixel_values=pixel_values,
            attention_mask=attention_mask,
            kv_cache=kv_cache,
        )
        kv_cache = outputs["kv_cache"]
        next_token_logits = outputs["logits"][:, -1, :]
        # Sample the next token
        if do_sample:
            # Apply temperature
            next_token_logits = torch.softmax(next_token_logits / temperature, dim=-1)
            next_token = _sample_top_p(next_token_logits, top_p)
        else:
            next_token = torch.argmax(next_token_logits, dim=-1, keepdim=True)
        assert next_token.size() == (1, 1)
        next_token = next_token.squeeze(0)  # Remove batch dimension
        generated_tokens.append(next_token)
        # Stop if the stop token has been generated
        if next_token.item() == stop_token:
            break
        # Append the next token to the input
        input_ids = next_token.unsqueeze(-1)
        attention_mask = torch.cat(
            [attention_mask, torch.ones((1, 1), device=input_ids.device)], dim=-1
        )

    end_time = time.time()
    latency = end_time - start_time

    generated_tokens = torch.cat(generated_tokens, dim=-1)
    # Decode the generated tokens
    decoded = processor.tokenizer.decode(generated_tokens, skip_special_tokens=True)

    print("Result!!!")
    print(prompt + decoded)
    print(f"InferenceLatency: {latency:.2f} seconds")

    # Get memory usage
    process = psutil.Process(os.getpid())
    memory_usage = process.memory_info().rss / 1024 / 1024 / 1024  # Convert to GB
    print(f"Memory Usage: {memory_usage:.2f} GB")

    if device == "cuda":
        gpu_memory = torch.cuda.max_memory_allocated() / 1024 / 1024  # Convert to GB
        print(f"GPU Memory Usage: {gpu_memory:.2f} GB")
        torch.cuda.reset_peak_memory_stats()


def _sample_top_p(probs: torch.Tensor, p: float):
    # (B, vocab_size)
    probs_sort, probs_idx = torch.sort(probs, dim=-1, descending=True)
    # (B, vocab_size)
    probs_sum = torch.cumsum(probs_sort, dim=-1)
    # (B, vocab_size)
     # (Substracting "probs_sort" shifts the cumulative sum by 1 position to the right before
masking)
    mask = probs_sum - probs_sort > p
    # Zero out all the probabilities of tokens that are not selected by the Top P
    probs_sort[mask] = 0.0
    # Redistribute the probabilities so that they sum up to 1.
    probs_sort.div_(probs_sort.sum(dim=-1, keepdim=True))
    # Sample a token (its index) from the top p distribution
    next_token = torch.multinomial(probs_sort, num_samples=1)
    # Get the token position in the vocabulary corresponding to the sampled index
    next_token = torch.gather(probs_idx, -1, next_token)
    return next_token



def main(
    model_path: str = None,
    prompt: str = None,
    image_file_path: str = None,
    max_tokens_to_generate: int = 100,
    temperature: float = 0.8,
    top_p: float = 0.9,
    do_sample: bool = False,
    only_cpu: bool = False,
):
    device = "cpu"

    if not only_cpu:
        if torch.cuda.is_available():
            device = "cuda"
        elif torch.backends.mps.is_available():
            device = "mps"

    print("Device in use: ", device)

    print(f"Loading model")
    start_time = time.time()
    model, tokenizer = load_hf_model(model_path, device)
    model = model.to(device).eval()

    num_image_tokens = model.config.vision_config.num_image_tokens
    image_size = model.config.vision_config.image_size
    processor = PaliGemmaProcessor(tokenizer, num_image_tokens, image_size)

    print(f"Model loaded in {time.time() - start_time:.2f} seconds")
    print("Running inference")
    with torch.no_grad():
        test_inference(
            model,
            processor,
            device,
            prompt,
            image_file_path,
            max_tokens_to_generate,
            temperature,
            top_p,
            do_sample,
        )


if __name__ == "__main__":
    main(
        model_path="paligemma-3b-pt-224/",
        prompt="this building is ",
        image_file_path="test_images/1.jpg",
        max_tokens_to_generate=100,
        temperature=0.8,
        top_p=0.9,
        do_sample=False,
        only_cpu=True,
    )
```

decoder_1.py

```
import torch
from torch import nn
from typing import Optional, Tuple, List
from torch.nn import CrossEntropyLoss
import math
from vision_transformer_1 import VisionConfig, VisionModel

class KVCache():
    """
    KVCache stores previous key and value states to avoid recomputing them during text generation.
    This makes generation more efficient by reusing previous attention computations.
    Think of it like the model's "memory" of what it has processed so far.

    During text generation, instead of recomputing attention for all previous tokens,
    we can reuse the cached key/value states from previous forward passes.
    This significantly speeds up the generation process.
    """

    def __init__(self) -> None:
        self.key_cache: List[torch.Tensor] = []
        self.value_cache: List[torch.Tensor] = []

    def num_items(self) -> int:
        if len(self.key_cache) == 0:
            return 0
        else:
            # The shape of the key_cache is [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
            return self.key_cache[0].shape[-2]

    def update(
        self,
        key_states: torch.Tensor,
        value_states: torch.Tensor,
        layer_idx: int,
    ) -> Tuple[torch.Tensor, torch.Tensor]:
        if len(self.key_cache) <= layer_idx:
            # If we never added anything to the KV-Cache of this layer, let's create it.
            self.key_cache.append(key_states)
            self.value_cache.append(value_states)
        else:
            # ... otherwise we concatenate the new keys with the existing ones.
            # each tensor has shape: [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
            self.key_cache[layer_idx] = torch.cat([self.key_cache[layer_idx], key_states], dim=-2)
              self.value_cache[layer_idx] = torch.cat([self.value_cache[layer_idx], value_states],
dim=-2)

        # ... and then we return all the existing keys + the new ones.
        return self.key_cache[layer_idx], self.value_cache[layer_idx]
```

decoder_1.py

```
class GemmaConfig():
    """
    Configuration class that stores all the hyperparameters needed for the Gemma model.
    This includes things like model size (hidden_size), number of layers, attention heads, etc.
    Think of it as a recipe card that defines how big and complex the model should be.
    """

    def __init__(
        self,
        vocab_size,
        hidden_size,
        intermediate_size,
        num_hidden_layers,
        num_attention_heads,
        num_key_value_heads,
        head_dim=256,
        max_position_embeddings=8192,
        rms_norm_eps=1e-6,
        rope_theta=10000.0,
        attention_bias=False,
        attention_dropout=0.0,
        pad_token_id=None,
        **kwargs,
    ):
        super().__init__()
        self.vocab_size = vocab_size
        self.max_position_embeddings = max_position_embeddings
        self.hidden_size = hidden_size
        self.intermediate_size = intermediate_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.head_dim = head_dim
        self.num_key_value_heads = num_key_value_heads
        self.rms_norm_eps = rms_norm_eps
        self.rope_theta = rope_theta
        self.attention_bias = attention_bias
        self.attention_dropout = attention_dropout
        self.pad_token_id = pad_token_id

class PaliGemmaConfig():

    def __init__(
        self,
        vision_config=None,
        text_config=None,
        ignore_index=-100,
        image_token_index=256000,
        vocab_size=257152,
        projection_dim=2048,
        hidden_size=2048,
        pad_token_id=None,
        **kwargs,
    ):
        super().__init__()
        self.ignore_index = ignore_index
        self.image_token_index = image_token_index
        self.vocab_size = vocab_size
        self.projection_dim = projection_dim
        self.hidden_size = hidden_size
        self.vision_config = vision_config
        self.is_encoder_decoder = False
        self.pad_token_id = pad_token_id

        self.vision_config = VisionConfig(**vision_config)
        self.text_config = text_config

        self.text_config = GemmaConfig(**text_config, pad_token_id=pad_token_id)
        self.vocab_size = self.text_config.vocab_size

        self.text_config.num_image_tokens = (self.vision_config.image_size // self.vision_config.patch_size) ** 2
        self.vision_config.projection_dim = projection_dim


class GemmaRMSNorm(nn.Module):
    """
    A special type of normalization layer used in Gemma (similar to LayerNorm but slightly different).
    Normalization helps keep the values in a reasonable range as they flow through the network.
    Without it, values could explode or vanish, making training impossible.
    """
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.zeros(dim))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x):
        output = self._norm(x.float())
        # Llama does x.to(float16) * w whilst Gemma is (x * w).to(float16)
        # See https://github.com/huggingface/transformers/pull/29402
        output = output * (1.0 + self.weight.float())
        return output.type_as(x)

class GemmaRotaryEmbedding(nn.Module):
    """
    Implements Rotary Position Embedding (RoPE) which helps the model understand token positions.
    Instead of adding position information directly, it rotates the token embeddings in a way that
    naturally represents their positions. This is like giving each token a unique "angle" based on
    where it appears in the sequence.
    """
    def __init__(self, dim, max_position_embeddings=2048, base=10000, device=None):
        super().__init__()

        self.dim = dim # it is set to the head_dim
        self.max_position_embeddings = max_position_embeddings
        self.base = base

        # Calculate the theta according to the formula theta_i = base^(2i/dim) where i = 0, 1, 2, ..., dim // 2
        inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2, dtype=torch.int64).float() / self.dim))
        self.register_buffer("inv_freq", tensor=inv_freq, persistent=False)

    @torch.no_grad()
    def forward(self, x, position_ids, seq_len=None):
        # x: [bs, num_attention_heads, seq_len, head_size]
        self.inv_freq.to(x.device)
        # Copy the inv_freq tensor for batch in the sequence
        # inv_freq_expanded: [Batch_Size, Head_Dim // 2, 1]
        inv_freq_expanded = self.inv_freq[None, :, None].float().expand(position_ids.shape[0], -1, 1)
        # position_ids_expanded: [Batch_Size, 1, Seq_Len]
        position_ids_expanded = position_ids[:, None, :].float()
        device_type = x.device.type
        device_type = device_type if isinstance(device_type, str) and device_type != "mps" else "cpu"
        with torch.autocast(device_type=device_type, enabled=False):
            # Multiply each theta by the position (which is the argument of the sin and cos functions)
            # freqs: [Batch_Size, Head_Dim // 2, 1] @ [Batch_Size, 1, Seq_Len] --> [Batch_Size, Seq_Len, Head_Dim // 2]
            freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(1, 2)
            # emb: [Batch_Size, Seq_Len, Head_Dim]
            emb = torch.cat((freqs, freqs), dim=-1)
            # cos, sin: [Batch_Size, Seq_Len, Head_Dim]
            cos = emb.cos()
            sin = emb.sin()
        return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)



def rotate_half(x):
    # Build the [-x2, x1, -x4, x3, ...] tensor for the sin part of the positional encoding.
    x1 = x[..., : x.shape[-1] // 2] # Takes the first half of the last dimension
    x2 = x[..., x.shape[-1] // 2 :] # Takes the second half of the last dimension
    return torch.cat((-x2, x1), dim=-1)



def apply_rotary_pos_emb(q, k, cos, sin, unsqueeze_dim=1):
    cos = cos.unsqueeze(unsqueeze_dim) # Add the head dimension
    sin = sin.unsqueeze(unsqueeze_dim) # Add the head dimension
    # Apply the formula (34) of the Rotary Positional Encoding paper.
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed


class GemmaMLP(nn.Module):
    """
    Multi-Layer Perceptron - a simple feed-forward neural network.
    After attention gathers information from other tokens, the MLP processes
    this information further. Think of it as the "thinking" step after the
    "gathering information" step of attention.
    """
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size
        self.gate_proj = nn.Linear(self.hidden_size, self.intermediate_size, bias=False)
        self.up_proj = nn.Linear(self.hidden_size, self.intermediate_size, bias=False)
        self.down_proj = nn.Linear(self.intermediate_size, self.hidden_size, bias=False)

    def forward(self, x):
        # Equivalent to:
        # y = self.gate_proj(x) # [Batch_Size, Seq_Len, Hidden_Size] -> [Batch_Size, Seq_Len,
Intermediate_Size]
        # y = torch.gelu(y, approximate="tanh") # [Batch_Size, Seq_Len, Intermediate_Size]
        # j = self.up_proj(x) # [Batch_Size, Seq_Len, Hidden_Size] -> [Batch_Size, Seq_Len,
Intermediate_Size]
        # z = y * j # [Batch_Size, Seq_Len, Intermediate_Size]
        # z = self.down_proj(z) # [Batch_Size, Seq_Len, Intermediate_Size] -> [Batch_Size,
Seq_Len, Hidden_Size]
        return self.down_proj(nn.functional.gelu(self.gate_proj(x), approximate="tanh") *
self.up_proj(x))

def repeat_kv(hidden_states: torch.Tensor, n_rep: int) -> torch.Tensor:
    batch, num_key_value_heads, slen, head_dim = hidden_states.shape
    if n_rep == 1:
        return hidden_states
    hidden_states = hidden_states[:, :, None, :, :].expand(batch, num_key_value_heads, n_rep,
slen, head_dim)
    return hidden_states.reshape(batch, num_key_value_heads * n_rep, slen, head_dim)

class GemmaAttention(nn.Module):
    """
    The attention mechanism - the heart of the transformer architecture.
    It lets each token "pay attention" to other tokens when processing the sequence.

    For example, in "The cat sat on the mat", when processing "sat",
    attention helps the model look at "cat" to know WHO sat, and "mat" to know WHERE.

    The attention mechanism works in three steps:
    1. Create queries (what to look for), keys (what to match against), and values (what
information to retrieve)
    2. Compute attention scores between queries and keys
    3. Use these scores to weight the values and create the final output

    This implementation also includes:
    - Multi-head attention (parallel attention computations)
    - Grouped-query attention (sharing key/value heads across query heads)
    - Rotary position embeddings (RoPE) for handling token positions
    - KV-caching for efficient text generation
    """
    def __init__(self, config: GemmaConfig, layer_idx: Optional[int] = None):
        super().__init__()
        self.config = config
        self.layer_idx = layer_idx

        self.attention_dropout = config.attention_dropout
        self.hidden_size = config.hidden_size
        self.num_heads = config.num_attention_heads
        self.head_dim = config.head_dim
        self.num_key_value_heads = config.num_key_value_heads
        self.num_key_value_groups = self.num_heads // self.num_key_value_heads
        self.max_position_embeddings = config.max_position_embeddings
        self.rope_theta = config.rope_theta
        self.is_causal = True

        assert self.hidden_size % self.num_heads == 0

        self.q_proj = nn.Linear(self.hidden_size, self.num_heads * self.head_dim, bias=config.attention_bias)
        self.k_proj = nn.Linear(self.hidden_size, self.num_key_value_heads * self.head_dim, bias=config.attention_bias)
        self.v_proj = nn.Linear(self.hidden_size, self.num_key_value_heads * self.head_dim, bias=config.attention_bias)
        self.o_proj = nn.Linear(self.num_heads * self.head_dim, self.hidden_size, bias=config.attention_bias)
        self.rotary_emb = GemmaRotaryEmbedding(
            self.head_dim,
            max_position_embeddings=self.max_position_embeddings,
            base=self.rope_theta,
        )

    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        kv_cache: Optional[KVCache] = None,
        **kwargs,
    ) -> Tuple[torch.Tensor, Optional[torch.Tensor], Optional[Tuple[torch.Tensor]]]:
        bsz, q_len, _ = hidden_states.size() # [Batch_Size, Seq_Len, Hidden_Size]
        # [Batch_Size, Seq_Len, Num_Heads_Q * Head_Dim]
        query_states = self.q_proj(hidden_states)
        # [Batch_Size, Seq_Len, Num_Heads_KV * Head_Dim]
        key_states = self.k_proj(hidden_states)
        # [Batch_Size, Seq_Len, Num_Heads_KV * Head_Dim]
        value_states = self.v_proj(hidden_states)
        # [Batch_Size, Num_Heads_Q, Seq_Len, Head_Dim]
        query_states = query_states.view(bsz, q_len, self.num_heads, self.head_dim).transpose(1,
2)
        # [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
        key_states = key_states.view(bsz, q_len, self.num_key_value_heads,
self.head_dim).transpose(1, 2)
        # [Batch_Size, Num_Heads_KV, Seq_Len, Head_Dim]
        value_states = value_states.view(bsz, q_len, self.num_key_value_heads,
self.head_dim).transpose(1, 2)

        # [Batch_Size, Seq_Len, Head_Dim], [Batch_Size, Seq_Len, Head_Dim]
        cos, sin = self.rotary_emb(value_states, position_ids, seq_len=None)
        # [Batch_Size, Num_Heads_Q, Seq_Len, Head_Dim], [Batch_Size, Num_Heads_KV, Seq_Len,
Head_Dim]
        query_states, key_states = apply_rotary_pos_emb(query_states, key_states, cos, sin)
        if kv_cache is not None:
            key_states, value_states = kv_cache.update(key_states, value_states, self.layer_idx)

        # Repeat the key and values to match the number of heads of the query
        key_states = repeat_kv(key_states, self.num_key_value_groups)
        value_states = repeat_kv(value_states, self.num_key_value_groups)
        # Perform the calculation as usual, Q * K^T / sqrt(head_dim). Shape: [Batch_Size,
Num_Heads_Q, Seq_Len_Q, Seq_Len_KV]
        attn_weights = torch.matmul(query_states, key_states.transpose(2, 3)) /
math.sqrt(self.head_dim)

        assert attention_mask is not None
        attn_weights = attn_weights + attention_mask

        # Apply the softmax
        # [Batch_Size, Num_Heads_Q, Seq_Len_Q, Seq_Len_KV]
        attn_weights = nn.functional.softmax(attn_weights, dim=-1,
dtype=torch.float32).to(query_states.dtype)
        # Apply the dropout
        attn_weights = nn.functional.dropout(attn_weights, p=self.attention_dropout,
training=self.training)
        # Multiply by the values. [Batch_Size, Num_Heads_Q, Seq_Len_Q, Seq_Len_KV] x [Batch_Size,
Num_Heads_KV, Seq_Len_KV, Head_Dim] -> [Batch_Size, Num_Heads_Q, Seq_Len_Q, Head_Dim]
        attn_output = torch.matmul(attn_weights, value_states)

        if attn_output.size() != (bsz, self.num_heads, q_len, self.head_dim):
            raise ValueError(
                f"`attn_output` should be of size {(bsz, self.num_heads, q_len, self.head_dim)},
but is"
                f" {attn_output.size()}"
            )
        # Make sure the sequence length is the second dimension. # [Batch_Size, Num_Heads_Q,
Seq_Len_Q, Head_Dim] -> [Batch_Size, Seq_Len_Q, Num_Heads_Q, Head_Dim]
        attn_output = attn_output.transpose(1, 2).contiguous()
        # Concatenate all the heads together. [Batch_Size, Seq_Len_Q, Num_Heads_Q, Head_Dim] ->
```

```
[Batch_Size, Seq_Len_Q, Num_Heads_Q * Head_Dim]
        attn_output = attn_output.view(bsz, q_len, -1)
        # Multiply by W_o. [Batch_Size, Seq_Len_Q, Hidden_Size]
        attn_output = self.o_proj(attn_output)

        return attn_output, attn_weights

class GemmaDecoderLayer(nn.Module):
    """
    One complete layer of the transformer decoder.
    Each layer does these steps:
    1. Self-attention: Look at other tokens to gather context
    2. Add & normalize: Add the original input back and normalize
    3. MLP: Process the gathered information
    4. Add & normalize again

    The model stacks many of these layers to build deep understanding.
    """
    def __init__(self, config: GemmaConfig, layer_idx: int):
        super().__init__()
        self.hidden_size = config.hidden_size

        self.self_attn = GemmaAttention(config=config, layer_idx=layer_idx)

        self.mlp = GemmaMLP(config)
        self.input_layernorm = GemmaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)
        self.post_attention_layernorm = GemmaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def forward(
        self,
        hidden_states: torch.Tensor,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        kv_cache: Optional[KVCache] = None,
    ) -> Tuple[torch.FloatTensor, Optional[Tuple[torch.FloatTensor, torch.FloatTensor]]]:
        residual = hidden_states
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.input_layernorm(hidden_states)

        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states, _, = self.self_attn(
            hidden_states=hidden_states,
            attention_mask=attention_mask,
            position_ids=position_ids,
            kv_cache=kv_cache,
        )
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = residual + hidden_states

        # [Batch_Size, Seq_Len, Hidden_Size]
        residual = hidden_states
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.post_attention_layernorm(hidden_states)
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.mlp(hidden_states)
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = residual + hidden_states

        return hidden_states

class GemmaModel(nn.Module):

    def __init__(self, config: GemmaConfig):
        super().__init__()
        self.config = config
        self.padding_idx = config.pad_token_id
        self.vocab_size = config.vocab_size

        self.embed_tokens = nn.Embedding(config.vocab_size, config.hidden_size, self.padding_idx)
        self.layers = nn.ModuleList(
                                     [GemmaDecoderLayer(config,   layer_idx)   for   layer_idx   in
range(config.num_hidden_layers)]
        )
        self.norm = GemmaRMSNorm(config.hidden_size, eps=config.rms_norm_eps)

    def get_input_embeddings(self):
        return self.embed_tokens

    # Ignore copy
    def forward(
        self,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        kv_cache: Optional[KVCache] = None,
    ) -> torch.FloatTensor:
        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = inputs_embeds
        # [Batch_Size, Seq_Len, Hidden_Size]
        normalizer = torch.tensor(self.config.hidden_size**0.5, dtype=hidden_states.dtype)
        hidden_states = hidden_states * normalizer

        for decoder_layer in self.layers:
            # [Batch_Size, Seq_Len, Hidden_Size]
            hidden_states = decoder_layer(
                hidden_states,
                attention_mask=attention_mask,
                position_ids=position_ids,
                kv_cache=kv_cache,
            )

        # [Batch_Size, Seq_Len, Hidden_Size]
        hidden_states = self.norm(hidden_states)
        # [Batch_Size, Seq_Len, Hidden_Size]
        return hidden_states

class GemmaForCausalLM(nn.Module):

    def __init__(self, config):
        super().__init__()
        self.config = config
        self.model = GemmaModel(config)
        self.vocab_size = config.vocab_size
        self.lm_head = nn.Linear(config.hidden_size, config.vocab_size, bias=False)

    def get_input_embeddings(self):
        return self.model.embed_tokens

    def tie_weights(self):
        self.lm_head.weight = self.model.embed_tokens.weight

    def forward(
        self,
        attention_mask: Optional[torch.Tensor] = None,
        position_ids: Optional[torch.LongTensor] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        kv_cache: Optional[KVCache] = None,
    ) -> Tuple:

        # input_embeds: [Batch_Size, Seq_Len, Hidden_Size]
        # outputs: [Batch_Size, Seq_Len, Hidden_Size]
        outputs = self.model(
            attention_mask=attention_mask,
            position_ids=position_ids,
            inputs_embeds=inputs_embeds,
            kv_cache=kv_cache,
        )

        hidden_states = outputs
        logits = self.lm_head(hidden_states)
        logits = logits.float()

        return_data = {
            "logits": logits,
        }

        if kv_cache is not None:
            # Return the updated cache
            return_data["kv_cache"] = kv_cache

        return return_data

class PaliGemmaMultiModalProjector(nn.Module):
    def __init__(self, config: PaliGemmaConfig):
        super().__init__()
```

```
                                   self.linear    =    nn.Linear(config.vision_config.hidden_size,
config.vision_config.projection_dim, bias=True)

    def forward(self, image_features):
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Projection_Dim]
        hidden_states = self.linear(image_features)
        return hidden_states

class PaliGemmaForConditionalGeneration(nn.Module):
    """
    The complete multimodal model that can process both images and text.
    It combines:
    1. A vision model to understand images
    2. A language model (Gemma) to understand and generate text
    3. A projector to connect image features to text features

    This allows the model to generate text based on both image and text inputs.

    The model works in several steps:
    1. Process images through the vision tower to extract visual features
    2. Project these visual features to match the text embedding space
    3. Combine image and text embeddings into a single sequence
    4. Process the combined sequence through the language model
    5. Generate text outputs based on both visual and textual context

    This architecture enables tasks like:
    - Image captioning
    - Visual question answering
    - Image-guided text generation
    """
    def __init__(self, config: PaliGemmaConfig):
        super().__init__()
        self.config = config
        self.vision_tower = VisionModel(config.vision_config)
        self.multi_modal_projector = PaliGemmaMultiModalProjector(config)
        self.vocab_size = config.vocab_size

        language_model = GemmaForCausalLM(config.text_config)
        self.language_model = language_model

         self.pad_token_id = self.config.pad_token_id if self.config.pad_token_id is not None else
-1

    def tie_weights(self):
        return self.language_model.tie_weights()

    def _merge_input_ids_with_image_features(
         self, image_features: torch.Tensor, inputs_embeds: torch.Tensor, input_ids: torch.Tensor,
attention_mask: torch.Tensor, kv_cache: Optional[KVCache] = None
    ):
        """
        This crucial method combines image and text features into one sequence.
        It's like creating a single story that weaves together what the model
        sees in the image and what it reads in the text.

        For example, if you have an image of a dog and text "The dog is",
        this method helps the model use both pieces of information to maybe
        complete the sentence with "playing in the park" based on what it
        sees in the image.
        """
        _, _, embed_dim = image_features.shape
        batch_size, sequence_length = input_ids.shape
        dtype, device = inputs_embeds.dtype, inputs_embeds.device
        # Shape: [Batch_Size, Seq_Len, Hidden_Size]
        scaled_image_features = image_features / (self.config.hidden_size**0.5)

        # Combine the embeddings of the image tokens, the text tokens and mask out all the padding
tokens.
        final_embedding = torch.zeros(batch_size, sequence_length, embed_dim,
dtype=inputs_embeds.dtype, device=inputs_embeds.device)
        # Shape: [Batch_Size, Seq_Len]. True for text tokens
        text_mask = (input_ids != self.config.image_token_index) & (input_ids !=
self.pad_token_id)
        # Shape: [Batch_Size, Seq_Len]. True for image tokens
        image_mask = input_ids == self.config.image_token_index
        # Shape: [Batch_Size, Seq_Len]. True for padding tokens
        pad_mask = input_ids == self.pad_token_id

        # We need to expand the masks to the embedding dimension otherwise we can't use them in
torch.where
        text_mask_expanded = text_mask.unsqueeze(-1).expand(-1, -1, embed_dim)
        pad_mask_expanded = pad_mask.unsqueeze(-1).expand(-1, -1, embed_dim)
        image_mask_expanded = image_mask.unsqueeze(-1).expand(-1, -1, embed_dim)

        # Add the text embeddings
        final_embedding = torch.where(text_mask_expanded, inputs_embeds, final_embedding)
        # Insert image embeddings. We can't use torch.where because the sequence length of
scaled_image_features is not equal to the sequence length of the final embedding
        final_embedding = final_embedding.masked_scatter(image_mask_expanded,
scaled_image_features)
        # Zero out padding tokens
        final_embedding = torch.where(pad_mask_expanded, torch.zeros_like(final_embedding),
final_embedding)

        #### CREATE THE ATTENTION MASK ####

        dtype, device = inputs_embeds.dtype, inputs_embeds.device
        min_dtype = torch.finfo(dtype).min
        q_len = inputs_embeds.shape[1]

        if kv_cache is None or kv_cache.num_items() == 0:
            # Do not mask any token, because we're in the prefill phase
            # This only works when we have no padding
            causal_mask = torch.full(
                (batch_size, q_len, q_len), fill_value=0, dtype=dtype, device=device
            )
        else:
            # Since we are generating tokens, the query must be one single token
            assert q_len == 1
            kv_len = kv_cache.num_items() + q_len
            # Also in this case we don't need to mask anything, since each query should be able to
attend all previous tokens.
            # This only works when we have no padding
            causal_mask = torch.full(
                (batch_size, q_len, kv_len), fill_value=0, dtype=dtype, device=device
            )
        # Add the head dimension
        # [Batch_Size, Q_Len, KV_Len] -> [Batch_Size, Num_Heads_Q, Q_Len, KV_Len]
        causal_mask = causal_mask.unsqueeze(1)

        if kv_cache is not None and kv_cache.num_items() > 0:
            # The position of the query is just the last position
            position_ids = attention_mask.cumsum(-1)[:, -1]
            if position_ids.dim() == 1:
                position_ids = position_ids.unsqueeze(0)
        else:
            # Create a position_ids based on the size of the attention_mask
            # For masked tokens, use the number 1 as position.
                  position_ids = (attention_mask.cumsum(-1)).masked_fill_((attention_mask == 0),
1).to(device)

        return final_embedding, causal_mask, position_ids

    def forward(
        self,
        input_ids: torch.LongTensor = None,
        pixel_values: torch.FloatTensor = None,
        attention_mask: Optional[torch.Tensor] = None,
        kv_cache: Optional[KVCache] = None,
    ) -> Tuple:

        # Make sure the input is right-padded
        assert torch.all(attention_mask == 1), "The input cannot be padded"

        # 1. Extra the input embeddings
        # shape: (Batch_Size, Seq_Len, Hidden_Size)
        inputs_embeds = self.language_model.get_input_embeddings()(input_ids)

        # 2. Merge text and images
        # [Batch_Size, Channels, Height, Width] -> [Batch_Size, Num_Patches, Embed_Dim]
        selected_image_feature = self.vision_tower(pixel_values.to(inputs_embeds.dtype))
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Hidden_Size]
        image_features = self.multi_modal_projector(selected_image_feature)

        # Merge the embeddings of the text tokens and the image tokens
                                        inputs_embeds,     attention_mask,     position_ids     =
self._merge_input_ids_with_image_features(image_features,         inputs_embeds,         input_ids,
attention_mask, kv_cache)

        outputs = self.language_model(
            attention_mask=attention_mask,
            position_ids=position_ids,
            inputs_embeds=inputs_embeds,
            kv_cache=kv_cache,
        )

        return outputs
```

text_image_token_processor_1.py

```python
from typing import Dict, List, Optional, Union, Tuple, Iterable
import numpy as np
from PIL import Image
import torch

# Standard normalization values used in many computer vision models
# These values help center the image data around 0 and scale it appropriately
IMAGENET_STANDARD_MEAN = [0.5, 0.5, 0.5]  # One value for each RGB channel
IMAGENET_STANDARD_STD = [0.5, 0.5, 0.5]   # One value for each RGB channel


def add_image_tokens_to_prompt(prefix_prompt, bos_token, image_seq_len, image_token):
    # Quoting from the blog (https://huggingface.co/blog/paligemma#detailed-inference-process):
    #   The input text is tokenized normally.
    #   A <bos> token is added at the beginning, and an additional newline token (\n) is appended.
    #   This newline token is an essential part of the input prompt the model was trained with, so adding it explicitly ensures it's always there.
    #   The tokenized text is also prefixed with a fixed number of <image> tokens.
    # NOTE: from the paper it looks like the `\n` should be tokenized separately, but in the HF implementation this is not done.
    #       ref to HF implementation: https://github.com/huggingface/transformers/blob/7f79a97399bb52aad8460e1da2f36577d5dccfed/src/transformers/models/paligemma/processing_paligemma.py#L55-L73
    return f"{image_token * image_seq_len}{bos_token}{prefix_prompt}\n"


def rescale(
    image: np.ndarray, scale: float, dtype: np.dtype = np.float32
) -> np.ndarray:
    rescaled_image = image * scale
    rescaled_image = rescaled_image.astype(dtype)
    return rescaled_image


def resize(
    image: Image,
    size: Tuple[int, int],
    resample: Image.Resampling = None,
    reducing_gap: Optional[int] = None,
) -> np.ndarray:
    height, width = size
    resized_image = image.resize(
        (width, height), resample=resample, reducing_gap=reducing_gap
    )
    return resized_image


def normalize(
    image: np.ndarray,
    mean: Union[float, Iterable[float]],
    std: Union[float, Iterable[float]],
) -> np.ndarray:
    mean = np.array(mean, dtype=image.dtype)
    std = np.array(std, dtype=image.dtype)
    image = (image - mean) / std
    return image


def process_images(
    images: List[Image.Image],
    size: Dict[str, int] = None,
    resample: Image.Resampling = None,
    rescale_factor: float = None,
    image_mean: Optional[Union[float, List[float]]] = None,
    image_std: Optional[Union[float, List[float]]] = None,
) -> List[np.ndarray]:
    """
    Main function that processes images through several steps to prepare them for the model:
    1. Resize to specified dimensions
    2. Convert to numpy arrays
    3. Rescale pixel values
    4. Normalize the values
    5. Rearrange dimensions to model's expected format
    """
    height, width = size[0], size[1]
    # Step 1: Resize all images to the same size
    images = [
        resize(image=image, size=(height, width), resample=resample) for image in images
    ]
    # Step 2: Convert PIL images to numpy arrays for numerical processing
    images = [np.array(image) for image in images]
    # Step 3: Rescale pixel values from [0, 255] to [0, 1] range
    images = [rescale(image, scale=rescale_factor) for image in images]
    # Step 4: Normalize images using mean and standard deviation
    images = [normalize(image, mean=image_mean, std=image_std) for image in images]
    # Step 5: Rearrange dimensions from [Height, Width, Channel] to [Channel, Height, Width]
    images = [image.transpose(2, 0, 1) for image in images]
    return images


class PaliGemmaProcessor:
    """
    A processor class that handles both image and text processing for the PaLI-GEMMA model.
    It prepares images and text in the specific format required by the model.
    """
    IMAGE_TOKEN = "<image>"  # Special token that represents image content in the text

    def __init__(self, tokenizer, num_image_tokens: int, image_size: int):
        super().__init__()

        self.image_seq_length = num_image_tokens  # Number of image tokens to use
        self.image_size = image_size  # Target size for image processing

        # Add special tokens that the model understands
        tokens_to_add = {"additional_special_tokens": [self.IMAGE_TOKEN]}
        tokenizer.add_special_tokens(tokens_to_add)

        # Add special tokens for object detection and segmentation tasks
        # These tokens help the model understand spatial information in images
        EXTRA_TOKENS = [
            f"<loc{i:04d}>" for i in range(1024)
        ]  # Location tokens for identifying object positions
        EXTRA_TOKENS += [
            f"<seg{i:03d}>" for i in range(128)
        ]  # Segmentation tokens for identifying object boundaries
        tokenizer.add_tokens(EXTRA_TOKENS)

        # Get the ID for the image token for later use
        self.image_token_id = tokenizer.convert_tokens_to_ids(self.IMAGE_TOKEN)

        # Disable automatic addition of special tokens - we'll handle this manually
        tokenizer.add_bos_token = False  # BOS = Beginning of Sequence
        tokenizer.add_eos_token = False  # EOS = End of Sequence

        self.tokenizer = tokenizer

    def __call__(
        self,
        text: List[str],
        images: List[Image.Image],
        padding: str = "longest",
        truncation: bool = True,
    ) -> dict:
        """
        Main processing function that:
        1. Processes the images into the correct format
        2. Prepares the text with special tokens
        3. Combines everything into the format the model expects
        """
        # Currently only supports processing one image-text pair at a time
             assert  len(images)  ==  1  and  len(text)  ==  1,  f"Received  {len(images)}  images  for
{len(text)} prompts."

        # Process the images into tensor format
        pixel_values = process_images(
            images,
            size=(self.image_size, self.image_size),
            resample=Image.Resampling.BICUBIC,  # High-quality image resizing method
            rescale_factor=1 / 255.0,  # Convert pixel values from [0, 255] to [0, 1]
            image_mean=IMAGENET_STANDARD_MEAN,
            image_std=IMAGENET_STANDARD_STD,
        )
        # Stack individual images into a batch
```

```
pixel_values = np.stack(pixel_values, axis=0)
# Convert to PyTorch tensor for model input
pixel_values = torch.tensor(pixel_values)

# Prepare text inputs by adding image tokens and other special tokens
input_strings = [
    add_image_tokens_to_prompt(
        prefix_prompt=prompt,
        bos_token=self.tokenizer.bos_token,
        image_seq_len=self.image_seq_length,
        image_token=self.IMAGE_TOKEN,
    )
    for prompt in text
]

# Convert text to token IDs and create attention masks
inputs = self.tokenizer(
    input_strings,
    return_tensors="pt",  # Return PyTorch tensors
    padding=padding,      # Pad shorter sequences to match longest one
    truncation=truncation,# Cut off text that's too long
)

# Combine image and text inputs into a single dictionary
return_data = {"pixel_values": pixel_values, **inputs}

return return_data
```

vision_transformer_1.py

```
from typing import Optional, Tuple
import torch
import torch.nn as nn

# This file implements a Vision Transformer (ViT) model that processes images by:
# 1. Splitting the image into patches
# 2. Converting each patch into an embedding
# 3. Adding positional embeddings
# 4. Processing through transformer layers
# 5. Outputting final image features

class VisionConfig:
    """Configuration class that stores all the hyperparameters needed for the vision model"""

    def __init__(
        self,
        hidden_size=768,  # Size of the embeddings used throughout the model
        intermediate_size=3072,  # Size of the intermediate layer in MLP
        num_hidden_layers=12,  # Number of transformer layers
        num_attention_heads=12,  # Number of attention heads in each transformer layer
        num_channels=3,  # Number of input image channels (3 for RGB)
        image_size=224,  # Input image size (224x224 pixels)
        patch_size=16,  # Size of each image patch (16x16 pixels)
        layer_norm_eps=1e-6,  # Small constant for numerical stability in layer norm
        attention_dropout=0.0,  # Dropout rate for attention
            num_image_tokens: int = None,   # Number of image tokens (patches) - calculated as
(image_size/patch_size)^2
        **kwargs
    ):
        super().__init__()

        self.hidden_size = hidden_size
        self.intermediate_size = intermediate_size
        self.num_hidden_layers = num_hidden_layers
        self.num_attention_heads = num_attention_heads
        self.num_channels = num_channels
        self.patch_size = patch_size
        self.image_size = image_size
        self.attention_dropout = attention_dropout
        self.layer_norm_eps = layer_norm_eps
        self.num_image_tokens = num_image_tokens



class VisionEmbeddings(nn.Module):
    """Converts input images into patch embeddings and adds positional embeddings"""

    def __init__(self, config: VisionConfig):
        """Initialize the vision embeddings module.
        Args:
            config (VisionConfig): Configuration object containing model parameters
        """
        super().__init__()
        self.config = config
        self.embed_dim = config.hidden_size
        self.image_size = config.image_size
        self.patch_size = config.patch_size

        # Conv2d layer that splits image into patches and projects them to embedding dimension
        # For example: 224x224 image with 16x16 patches -> 14x14=196 patches
        self.patch_embedding = nn.Conv2d(
            in_channels=config.num_channels,
            out_channels=self.embed_dim,
            kernel_size=self.patch_size,
            stride=self.patch_size,
            padding="valid", # This indicates no padding is added
        )

        # Calculate total number of patches (e.g., 196 for 224x224 image with 16x16 patches)
        self.num_patches = (self.image_size // self.patch_size) ** 2
        self.num_positions = self.num_patches

        # Create learnable position embeddings for each patch
        self.position_embedding = nn.Embedding(self.num_positions, self.embed_dim)

        # Create fixed position IDs tensor [0, 1, 2, ..., num_patches-1]
        self.register_buffer(
            "position_ids",
            torch.arange(self.num_positions).expand((1, -1)),
            persistent=False,
        )

    def forward(self, pixel_values: torch.FloatTensor) -> torch.Tensor:
        """Convert input images into patch embeddings with positional encoding.

        Args:
            pixel_values (torch.FloatTensor): Input images of shape [Batch_Size, Channels, Height,
Width]

        Returns:
            torch.Tensor: Patch embeddings with positional encoding of shape [Batch_Size,
Num_Patches, Embed_Dim]
        """
        _, _, height, width = pixel_values.shape # [Batch_Size, Channels, Height, Width]
        # Convolve the `patch_size` kernel over the image, with no overlapping patches since the
stride is equal to the kernel size
        # The output of the convolution will have shape [Batch_Size, Embed_Dim, Num_Patches_H,
Num_Patches_W]
        # where Num_Patches_H = height // patch_size and Num_Patches_W = width // patch_size
        patch_embeds = self.patch_embedding(pixel_values)
        # [Batch_Size, Embed_Dim, Num_Patches_H, Num_Patches_W] -> [Batch_Size, Embed_Dim,
Num_Patches]
        # where Num_Patches = Num_Patches_H * Num_Patches_W
        embeddings = patch_embeds.flatten(2)
        # [Batch_Size, Embed_Dim, Num_Patches] -> [Batch_Size, Num_Patches, Embed_Dim]
        embeddings = embeddings.transpose(1, 2)
          # Add position embeddings to each patch. Each positional encoding is a vector of size
[Embed_Dim]
        # This helps the model understand the relative position of patches
        embeddings = embeddings + self.position_embedding(self.position_ids)
        # [Batch_Size, Num_Patches, Embed_Dim]
        return embeddings



class Attention(nn.Module):
     """Multi-headed attention mechanism that allows the model to focus on different parts of the
input"""

    def __init__(self, config):
        """Initialize multi-headed attention module.

        Args:
            config: Configuration object containing attention parameters
        """
        super().__init__()
        self.config = config
        self.embed_dim = config.hidden_size
        self.num_heads = config.num_attention_heads
        # Split embedding dimension among heads (e.g., 768/12 = 64)
        self.head_dim = self.embed_dim // self.num_heads
        # Scaling factor for dot product attention
        self.scale = self.head_dim**-0.5 # Equivalent to 1 / sqrt(self.head_dim)
        self.dropout = config.attention_dropout

        # Linear projections for Query, Key, Value
        self.k_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.v_proj = nn.Linear(self.embed_dim, self.embed_dim)
        self.q_proj = nn.Linear(self.embed_dim, self.embed_dim)
        # Final output projection
        self.out_proj = nn.Linear(self.embed_dim, self.embed_dim)

    def forward(
        self,
        hidden_states: torch.Tensor,
    ) -> Tuple[torch.Tensor, Optional[torch.Tensor]]:
        """Apply multi-headed self-attention to the input.

        Args:
                    hidden_states (torch.Tensor): Input tensor of shape [Batch_Size, Num_Patches,
Embed_Dim]

        Returns:
            Tuple[torch.Tensor, Optional[torch.Tensor]]:
                - Attention output of shape [Batch_Size, Num_Patches, Embed_Dim]
                - Attention weights of shape [Batch_Size, Num_Heads, Num_Patches, Num_Patches]
        """
        # hidden_states: [Batch_Size, Num_Patches, Embed_Dim]
        batch_size, seq_len, _ = hidden_states.size()
        # Project input into Query, Key, Value vectors
        # query_states: [Batch_Size, Num_Patches, Embed_Dim]
        query_states = self.q_proj(hidden_states)
        # key_states: [Batch_Size, Num_Patches, Embed_Dim]
        key_states = self.k_proj(hidden_states)
        # value_states: [Batch_Size, Num_Patches, Embed_Dim]
        value_states = self.v_proj(hidden_states)

        # Reshape Q, K, V to separate the heads
        # query_states: [Batch_Size, Num_Heads, Num_Patches, Head_Dim]
                      query_states   =   query_states.view(batch_size,   seq_len,   self.num_heads,
self.head_dim).transpose(1, 2)

                         key_states   =   key_states.view(batch_size,   seq_len,   self.num_heads,
self.head_dim).transpose(1, 2)

                      value_states   =   value_states.view(batch_size,   seq_len,   self.num_heads,
self.head_dim).transpose(1, 2)

        # Compute attention scores using scaled dot-product attention
             #  Calculate  the  attention  using  the  formula  Q  *  K^T  /  sqrt(d_k).  attn_weights:
[Batch_Size, Num_Heads, Num_Patches, Num_Patches]
        attn_weights = (torch.matmul(query_states, key_states.transpose(2, 3)) * self.scale)

        if attn_weights.size() != (batch_size, self.num_heads, seq_len, seq_len):
            raise ValueError(
                    f"Attention weights should be of size {(batch_size, self.num_heads, seq_len,
seq_len)}, but is"
                f" {attn_weights.size()}"
            )

        # Apply softmax to get attention probabilities
             #  Apply  the  softmax  row-wise.  attn_weights:  [Batch_Size,  Num_Heads,  Num_Patches,
Num_Patches]
                               attn_weights    =    nn.functional.softmax(attn_weights,    dim=-1,
dtype=torch.float32).to(query_states.dtype)
        # Apply dropout only during training
        attn_weights = nn.functional.dropout(attn_weights, p=self.dropout, training=self.training)
        # Multiply attention weights with values to get the final attention output
        # Multiply the attention weights by the value states. attn_output: [Batch_Size, Num_Heads,
Num_Patches, Head_Dim]
        attn_output = torch.matmul(attn_weights, value_states)

        if attn_output.size() != (batch_size, self.num_heads, seq_len, self.head_dim):
            raise ValueError(
                       f"`attn_output` should be of size {(batch_size, self.num_heads, seq_len,
self.head_dim)}, but is"
```

```
                f" {attn_output.size()}"
            )
        # Reshape output back to original dimensions
         # [Batch_Size, Num_Heads, Num_Patches, Head_Dim] -> [Batch_Size, Num_Patches, Num_Heads,
Head_Dim]
        attn_output = attn_output.transpose(1, 2).contiguous()
        # [Batch_Size, Num_Patches, Num_Heads, Head_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        attn_output = attn_output.reshape(batch_size, seq_len, self.embed_dim)
        # Final projection to combine all heads
        # [Batch_Size, Num_Patches, Embed_Dim]
        attn_output = self.out_proj(attn_output)

        return attn_output, attn_weights



class MLP(nn.Module):
    """Multi-Layer Perceptron that processes each patch independently"""

    def __init__(self, config):
        """Initialize MLP module.

        Args:
            config: Configuration object containing MLP parameters
        """
        super().__init__()
        self.config = config
        # First fully connected layer expands dimension
        self.fc1 = nn.Linear(config.hidden_size, config.intermediate_size)
        # Second fully connected layer projects back to original dimension
        self.fc2 = nn.Linear(config.intermediate_size, config.hidden_size)

    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        """Apply MLP transformation to input features.

        Args:
                  hidden_states  (torch.Tensor):  Input  tensor  of  shape  [Batch_Size,  Num_Patches,
Embed_Dim]

        Returns:
            torch.Tensor: Transformed features of shape [Batch_Size, Num_Patches, Embed_Dim]
        """
        # Expand dimension and apply GELU activation
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Intermediate_Size]
        hidden_states = self.fc1(hidden_states)
        # hidden_states: [Batch_Size, Num_Patches, Intermediate_Size]
        hidden_states = nn.functional.gelu(hidden_states, approximate="tanh")
        # Project back to original dimension
        # [Batch_Size, Num_Patches, Intermediate_Size] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = self.fc2(hidden_states)

        return hidden_states
```

```
class EncoderLayer(nn.Module):
    """Single transformer layer combining attention and MLP with residual connections"""

    def __init__(self, config: VisionConfig):
        """Initialize a single transformer encoder layer.

        Args:
            config (VisionConfig): Configuration object containing model parameters
        """
        super().__init__()
        self.embed_dim = config.hidden_size
        # Multi-head self attention layer
        self.self_attn = Attention(config)
        # Layer normalization before attention
        self.layer_norm1 = nn.LayerNorm(self.embed_dim, eps=config.layer_norm_eps)
        # MLP block
        self.mlp = MLP(config)
        # Layer normalization before MLP
        self.layer_norm2 = nn.LayerNorm(self.embed_dim, eps=config.layer_norm_eps)

    # Ignore copy
    def forward(
        self,
        hidden_states: torch.Tensor
    ) -> torch.Tensor:
        """Process input through self-attention and MLP with residual connections.

        Args:
                  hidden_states (torch.Tensor): Input tensor of shape [Batch_Size, Num_Patches,
Embed_Dim]

        Returns:
            torch.Tensor: Processed features of shape [Batch_Size, Num_Patches, Embed_Dim]
        """
        # First residual block: Self-attention
        # Save input for residual connection
        # residual: [Batch_Size, Num_Patches, Embed_Dim]
        residual = hidden_states
        # Layer norm before attention
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = self.layer_norm1(hidden_states)
        # Apply self-attention
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states, _ = self.self_attn(hidden_states=hidden_states)
        # Add residual connection
        # [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = residual + hidden_states

        # Second residual block: MLP
        # Save input for residual connection
        # residual: [Batch_Size, Num_Patches, Embed_Dim]
        residual = hidden_states
        # Layer norm before MLP
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = self.layer_norm2(hidden_states)
        # Apply MLP
        # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = self.mlp(hidden_states)
        # Add residual connection
        # [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = residual + hidden_states

        return hidden_states


class Encoder(nn.Module):
    """Stack of transformer encoder layers"""

    def __init__(self, config: VisionConfig):
        """Initialize the transformer encoder with multiple layers.

        Args:
            config (VisionConfig): Configuration object containing model parameters
        """
        super().__init__()
        self.config = config
        # Create list of encoder layers
        self.layers = nn.ModuleList(
            [EncoderLayer(config) for _ in range(config.num_hidden_layers)]
        )

    # Ignore copy
    def forward(
        self,
        inputs_embeds: torch.Tensor
    ) -> torch.Tensor:
        """Process input through all encoder layers sequentially.

        Args:
                inputs_embeds (torch.Tensor): Input embeddings of shape [Batch_Size, Num_Patches,
Embed_Dim]

        Returns:
            torch.Tensor: Encoded features of shape [Batch_Size, Num_Patches, Embed_Dim]
        """
        # Process input through each encoder layer sequentially
        # inputs_embeds: [Batch_Size, Num_Patches, Embed_Dim]
        hidden_states = inputs_embeds

        for encoder_layer in self.layers:
            # [Batch_Size, Num_Patches, Embed_Dim] -> [Batch_Size, Num_Patches, Embed_Dim]
            hidden_states = encoder_layer(hidden_states)
        return hidden_states


class VisionTransformer(nn.Module):
    """Main vision transformer model combining embeddings, encoder, and final layer norm"""

    def __init__(self, config: VisionConfig):
        """Initialize the complete vision transformer model.

        Args:
            config (VisionConfig): Configuration object containing model parameters
        """
        super().__init__()
        self.config = config
        embed_dim = config.hidden_size

        # Create embedding layer for converting image patches to embeddings
        self.embeddings = VisionEmbeddings(config)
        # Create encoder with multiple transformer layers
        self.encoder = Encoder(config)
        # Final layer normalization
        self.post_layernorm = nn.LayerNorm(embed_dim, eps=config.layer_norm_eps)

    def forward(self, pixel_values: torch.Tensor) -> torch.Tensor:
        """Process images through the complete vision transformer pipeline.

        Args:
            pixel_values (torch.Tensor): Input images of shape [Batch_Size, Channels, Height, 
Width]

        Returns:
            torch.Tensor: Final encoded features of shape [Batch_Size, Num_Patches, Embed_Dim]
        """
        # Convert image to patch embeddings
        # pixel_values: [Batch_Size, Channels, Height, Width] -> [Batch_Size, Num_Patches, 
Embed_Dim]
        hidden_states = self.embeddings(pixel_values)

        # Process through transformer encoder
        last_hidden_state = self.encoder(inputs_embeds=hidden_states)

        # Final layer normalization
        last_hidden_state = self.post_layernorm(last_hidden_state)

        return last_hidden_state


class VisionModel(nn.Module):
    """Wrapper class for the vision transformer model"""

    def __init__(self, config: VisionConfig):
        """Initialize the vision model wrapper.
        Args:
            config (VisionConfig): Configuration object containing model parameters
        """
        super().__init__()
        self.config = config
        self.vision_model = VisionTransformer(config)

    def forward(self, pixel_values) -> Tuple:
        """Process images through the vision transformer model.

        Args:
               pixel_values (torch.Tensor): Input images of shape [Batch_Size, Channels, Height,
Width]

        Returns:
            torch.Tensor: Encoded image features of shape [Batch_Size, Num_Patches, Embed_Dim]
        """
        # Process image through vision transformer
        # [Batch_Size, Channels, Height, Width] -> [Batch_Size, Num_Patches, Embed_Dim]
        return self.vision_model(pixel_values=pixel_values)
```