```python
## Building and training a bigram language model
from functools import partial
import math

import torch
import torch.nn as nn
from einops import einsum, reduce, rearrange

from config import BigramConfig, MiniGPTConfig

class BigramLanguageModel(nn.Module):
    """
    Class definition for a simple bigram language model.
    """

    def __init__(self, config):
        """
        Initialize the bigram language model with the given configuration.

        Args:
        config : BigramConfig (Defined in config.py)
            Configuration object containing the model parameters.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.embeddings)
           You can use the Embedding layer from PyTorch.
        2. A linear layer that maps embeddings to logits. (self.linear) **set bias to True**
        3. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        super().__init__()
        # ========= TODO : START ========= #

        self.embeddings = nn.Embedding(config.vocab_size, config.embed_dim )
        self.linear = nn.Linear(config.context_length*config.embed_dim, config.vocab_size ,
bias=True)
        self.dropout = nn.Dropout(p=config.dropout)

        # ========= TODO : END ========= #

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the bigram language model.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, 1) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, vocab_size) containing the logits.
        """

        # ========= TODO : START ========= #

        x = self.embeddings(x)
        x = self.linear(x)
        x = self.dropout(x)
        return x

        # ========= TODO : END ========= #
```

```python
    def _init_weights(self, module):
        """
        Weight initialization for better convergence.

        NOTE : You do not need to modify this function.
        """

        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.
        We will perform multinomial sampling which is very similar to greedy sampling,
        but instead of taking the token with the highest probability, we sample the next token
from a multinomial distribution.

        Remember in Bigram Language Model, we are only using the last token to predict the next
token.
        You should sample the next token x_t from the distribution p(x_t | x_{t-1}).

        Args:
        context : List[int]
            A list of integers (tokens) representing the context.
        max_new_tokens : int
            The maximum number of new tokens to generate.

        Output:
        List[int]
            A list of integers (tokens) representing the generated tokens.
        """

        ### ========= TODO : START ========= ###

        device = next(self.parameters()).device

        # Convert to list if needed
        if isinstance(context, torch.Tensor):
            context = context.tolist()

        for _ in range(max_new_tokens):
            last_token = torch.tensor([context[-1]], dtype=torch.long,
device=device).unsqueeze(0)  # (1, 1)
            logits = self(last_token)  # (1, 1, vocab_size)
            logits = logits[:, -1, :]  # (1, vocab_size)
            probs = torch.softmax(logits, dim=-1)  # (1, vocab_size)
            next_token = torch.multinomial(probs, num_samples=1)  # (1, 1)
            context.append(next_token.item())

        return torch.tensor(context, dtype=torch.long, device=device)

        ### ========= TODO : END ========= ###


class SingleHeadAttention(nn.Module):
    """
    Class definition for Single Head Causal Self Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)

    """

    def __init__(
```

```python
        self,
        input_dim,
        output_key_query_dim=None,
        output_value_dim=None,
        dropout=0.1,
        max_len=512,
    ):
        """
        Initialize the Single Head Attention Layer.

        The model should have the following layers:
        1. A linear layer for key. (self.key) **set bias to False**
        2. A linear layer for query. (self.query) **set bias to False**
        3. A linear layer for value. (self.value) # **set bias to False**
        4. A dropout layer. (self.dropout)
        5. A causal mask. (self.causal_mask) This should be registered as a buffer.
           - You can use the torch.tril function to create a lower triangular matrix.
           - In the skeleton we use register_buffer to register the causal mask as a buffer.
             This is typically used to register a buffer that should not to be considered a
model parameter.

        NOTE : Please make sure that the causal mask is upper triangular and not lower
triangular (this helps in setting up the test cases, )
        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        if output_key_query_dim:
            self.output_key_query_dim = output_key_query_dim
        else:
            self.output_key_query_dim = input_dim

        if output_value_dim:
            self.output_value_dim = output_value_dim
        else:
            self.output_value_dim = input_dim

        causal_mask = None  # You have to implement this, currently just a placeholder

        # ========= TODO : START ========= #

        self.key = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.query = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.value = nn.Linear(input_dim, self.output_value_dim, bias=False)

        self.dropout = nn.Dropout(dropout)

        mask = torch.triu(torch.ones(max_len, max_len), diagonal=1)
        causal_mask = mask.bool()

        # ========= TODO : END ========= #

        self.register_buffer(
            "causal_mask", causal_mask
        )  # Registering as buffer to avoid backpropagation

    def forward(self, x):
        """
        Forward pass of the Single Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
```

```python
            A tensor of shape (batch_size, num_tokens, output_value_dim) containing the output
tokens.

        Hint:
        - You need to 'trim' the causal mask to the size of the input tensor.
        """

        # ========= TODO : START ========= #

        B, T, _ = x.shape

        k = self.key(x)
        q = self.query(x)
        v = self.value(x)


        attn_scores = q @ k.transpose(-2, -1)  # (B, T, T)
        attn_scores = attn_scores / (self.output_key_query_dim ** 0.5)

        attn_scores = attn_scores.masked_fill(self.causal_mask[:T, :T], float('-inf'))

        attn_weights = torch.softmax(attn_scores, dim=-1)  # (B, T, T)
        attn_weights = self.dropout(attn_weights)

        output = attn_weights @ v  # (B, T, D_v)

        return output


        # ========= TODO : END ========= #


class MultiHeadAttention(nn.Module):
    """
    Class definition for Multi Head Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)
    """

    def __init__(self, input_dim, num_heads, dropout=0.1) -> None:
        """
        Initialize the Multi Head Attention Layer.

        The model should have the following layers:
        1. Multiple SingleHeadAttention layers. (self.head_{i}) Use setattr to dynamically set
the layers.
        2. A linear layer for output. (self.out) **set bias to True**
        3. A dropout layer. (self.dropout) Apply dropout to the output of the out layer.

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads

        # ========= TODO : START ========= #

        # self.head_{i} = ... # Use setattr to implement this dynamically, this is used as a
placeholder
        # self.out = ...
        # self.dropout = ...

        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        head_dim = input_dim // num_heads

        # Create and register each single-head attention layer as head_0, head_1, ..., head_{n}
```

```python
        for i in range(num_heads):
            setattr(
                self,
                f"head_{i}",
                SingleHeadAttention(
                    input_dim=input_dim,
                    output_key_query_dim=head_dim,
                    output_value_dim=head_dim,
                    dropout=dropout
                )
            )

        self.out = nn.Linear(input_dim, input_dim, bias=True)  # as required
        self.dropout = nn.Dropout(dropout)  # as required

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Multi Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        # ========= TODO : START ========= #

        # Collect outputs from each head
        head_outputs = []
        for i in range(self.num_heads):
            head = getattr(self, f"head_{i}")  # retrieve head_i
            head_output = head(x)  # (B, T, head_dim)
            head_outputs.append(head_output)

        # Concatenate all head outputs along the last dimension
        concat_output = torch.cat(head_outputs, dim=-1)  # (B, T, input_dim)

        # Final projection and dropout
        output = self.out(concat_output)  # (B, T, input_dim)
        output = self.dropout(output)

        return output
        # ========= TODO : END ========= #


class FeedForwardLayer(nn.Module):
    """
    Class definition for Feed Forward Layer.
    """

    def __init__(self, input_dim, feedforward_dim=None, dropout=0.1):
        """
        Initialize the Feed Forward Layer.

        The model should have the following layers:
        1. A linear layer for the feedforward network. (self.fc1) **set bias to True**
        2. A GELU activation function. (self.activation)
        3. A linear layer for the feedforward network. (self.fc2) ** set bias to True**
        4. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
```

```python
        """
        super().__init__()

        if feedforward_dim is None:
            feedforward_dim = input_dim * 4

        # ========= TODO : START ========= #

        self.fc1 = nn.Linear(input_dim, feedforward_dim, bias=True)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(feedforward_dim, input_dim, bias=True)
        self.dropout = nn.Dropout(dropout)



        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Feed Forward Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output 
tokens.
        """

        ### ========= TODO : START ========= ###

        out = self.fc1(x)  # (B, T, feedforward_dim)
        out = self.activation(out)  # (B, T, feedforward_dim)
        out = self.fc2(out)  # (B, T, input_dim)
        out = self.dropout(out)  # (B, T, input_dim)
        return out

        ### ========= TODO : END ========= ###


class LayerNorm(nn.Module):
    """
    LayerNorm module as in the paper https://arxiv.org/abs/1607.06450

    Note : Variance computation is done with biased variance.

    Hint :
    - You can use torch.var and specify whether to use biased variance or not.
    """

    def __init__(self, normalized_shape, eps=1e-05, elementwise_affine=True) -> None:
        super().__init__()

        self.normalized_shape = (normalized_shape,)
        self.eps = eps
        self.elementwise_affine = elementwise_affine

        if elementwise_affine:
            self.gamma = nn.Parameter(torch.ones(tuple(self.normalized_shape)))
            self.beta = nn.Parameter(torch.zeros(tuple(self.normalized_shape)))

    def forward(self, input):
        """
        Forward pass of the LayerNorm Layer.
```

```python
        Args:
        input : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        mean = None
        var = None
        # ========= TODO : START ========= #

        # Compute mean and variance
        mean = input.mean(dim=-1, keepdim=True)  # (B, T, 1)
        var = input.var(dim=-1, keepdim=True, unbiased=False)  # (B, T, 1)
        # Reshape mean and var to match the input shape
        mean = mean.expand_as(input)  # (B, T, D)
        var = var.expand_as(input)  # (B, T, D)
        # ========= TODO : END ========= #

        if self.elementwise_affine:
            return (
                self.gamma * (input - mean) / torch.sqrt((var + self.eps)) + self.beta
            )
        else:
            return (input - mean) / torch.sqrt((var + self.eps))


class TransformerLayer(nn.Module):
    """
    Class definition for a single transformer layer.
    """

    def __init__(self, input_dim, num_heads, feedforward_dim=None):
        super().__init__()
        """
        Initialize the Transformer Layer.
        We will use prenorm layer where we normalize the input before applying the attention
and feedforward layers.

        The model should have the following layers:
        1. A LayerNorm layer. (self.norm1)
        2. A MultiHeadAttention layer. (self.attention)
        3. A LayerNorm layer. (self.norm2)
        4. A FeedForwardLayer layer. (self.feedforward)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        # ========= TODO : START ========= #

        self.norm1 = LayerNorm(input_dim)
        self.attention = MultiHeadAttention(
            input_dim=input_dim, num_heads=num_heads
        )
        self.norm2 = LayerNorm(input_dim)
        self.feedforward = FeedForwardLayer(
            input_dim=input_dim, feedforward_dim=feedforward_dim
        )

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Transformer Layer.
```

```python
        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output
tokens.
        """

        # ========= TODO : START ========= #

        # LayerNorm + MultiHeadAttention
        x_new = self.norm1(x)  # (B, T, D)
        x_new = self.attention(x_new)  # (B, T, D)
        # Residual connection
        x = x + x_new  # (B, T, D)
        # LayerNorm + FeedForwardLayer
        x_new = self.norm2(x)  # (B, T, D)
        x_new = self.feedforward(x_new)  # (B, T, D)
        # Residual connection
        x = x + x_new  # (B, T, D)
        return x

        # ========= TODO : END ========= #


class MiniGPT(nn.Module):
    """
    Putting it all together: GPT model
    """

    def __init__(self, config) -> None:
        super().__init__()
        """
        Putting it all together: our own GPT model!

        Initialize the MiniGPT model.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.vocab_embedding)
        2. A positional embedding layer. (self.positional_embedding) We will use learnt
positional embeddings.
        3. A dropout layer for embeddings. (self.embed_dropout)
        4. Multiple TransformerLayer layers. (self.transformer_layers)
        5. A LayerNorm layer before the final layer. (self.prehead_norm)
        6. Final language Modelling head layer. (self.head) We will use weight tying
(https://paperswithcode.com/method/weight-tying) and set the weights of the head layer to be
the same as the vocab_embedding layer.

        NOTE: You do not need to modify anything here.
        """

        self.vocab_embedding = nn.Embedding(config.vocab_size, config.embed_dim)
        self.positional_embedding = nn.Embedding(
            config.context_length, config.embed_dim
        )
        self.embed_dropout = nn.Dropout(config.embed_dropout)

        self.transformer_layers = nn.ModuleList(
            [
                TransformerLayer(
                    config.embed_dim, config.num_heads, config.feedforward_size
                )
                for _ in range(config.num_layers)
            ]
```

```python
        )

        # prehead layer norm
        self.prehead_norm = LayerNorm(config.embed_dim)

        self.head = nn.Linear(
            config.embed_dim, config.vocab_size
        )  # Language modelling head

        if config.weight_tie:
            self.head.weight = self.vocab_embedding.weight

        # precreate positional indices for the positional embedding
        pos = torch.arange(0, config.context_length, dtype=torch.long)
        self.register_buffer("pos", pos, persistent=False)

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the MiniGPT model.

        Remember to add the positional embeddings to your input token!!

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, seq_len) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, seq_len, vocab_size) containing the logits.

        Hint:
        - You may need to 'trim' the positional embedding to match the input sequence length
        """

        ### ========= TODO : START ========= ###

        # Get the batch size and sequence length
        B, T = x.shape
        # Get the positional embeddings
        pos = self.pos[:T]

        # Get the token embeddings
        token_embeddings = self.vocab_embedding(x)

        # Get the positional embeddings
        pos_embeddings = self.positional_embedding(pos)
        # Add the token and positional embeddings
        x = token_embeddings + pos_embeddings
        # Apply dropout
        x = self.embed_dropout(x)
        # Pass through the transformer layers
        for layer in self.transformer_layers:
            x = layer(x)
        # Apply layer norm before the final layer
        x = self.prehead_norm(x)
        # Pass through the final layer
        x = self.head(x)
        # Return the logits
        return x

        ### ========= TODO : END ========= ###

    def _init_weights(self, module):
        """
        Weight initialization for better convergence.
```

```
        NOTE : You do not need to modify this function.
        """

        if isinstance(module, nn.Linear):
            if module._get_name() == "fc2":
                # GPT-2 style FFN init
                torch.nn.init.normal_(
                    module.weight, mean=0.0, std=0.02 / math.sqrt(2 * self.num_layers)
                )
            else:
                torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.

        Hint:
        - This should be similar to the Bigram Language Model, but you will use the entire
context to predict the next token.
          Instead of sampling from the distribution p(x_t | x_{t-1}),
            you will sample from the distribution p(x_t | x_{t-1}, x_{t-2}, ..., x_{t-n}),
            where n is the context length.
        - When decoding for the next token, you should use the logits of the last token in the
input sequence.

        """

        ### ========= TODO : START ========= ###

        # Move the context to the same device as the model
        context = context.to(self.pos.device)
        # Get the batch size and sequence length
        context = context.unsqueeze(0)
        B, T = context.shape
        # Create a tensor to hold the generated tokens
        generated_tokens = torch.zeros(
            (B, max_new_tokens), dtype=torch.long, device=self.pos.device
        )
        # Fill the generated tokens with the context
        generated_tokens[:, :T] = context
        # Generate new tokens
        for i in range(T,max_new_tokens):
            # Get the logits for the current context
            logits = self(context)
            # Get the last token's logits
            last_token_logits = logits[:, -1, :]
            # Sample from the distribution
            next_token = torch.multinomial(
                torch.softmax(last_token_logits, dim=-1), num_samples=1
            )
            # Add the new token to the generated tokens
            generated_tokens[:, i] = next_token.squeeze(1)
            # Update the context with the new token
            context = torch.cat((context, next_token), dim=1)
            # Update the batch size and sequence length
            B, T = context.shape
            # Trim the context to the maximum length
            context = context[:, -self.pos.shape[0] :]
        # Return the generated tokens
        return generated_tokens
```

```python
        ### ========= TODO : END ========= ###

class MultiQueryAttention(nn.Module):
    '''
    - Each head has its own Q
    - All heads share one K and one V
    '''

    def __int__(self, input_dim, num_heads, dropout=0.1) -> None:
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads

        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        self.head_dim = input_dim // num_heads
        self.scale = 1.0/math.sqrt(self.head_dim)

        self.q = nn.Linear(input_dim, input_dim, bias=False)
        self.kv = nn.Linear(input_dim, 2* self.head_dim, bias= False)

        self.out = nn.Linear(input_dim, input_dim, bias=True)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, _ = x.shape

        q = self.q(x).view(B,T,self.num_heads,self.head_dim)
        q = q.transpose(1,2)

        kv = self.kv(x)
        k, v = kv.split(self.head_dim, dim=-1)
        k = k.unsqueeze(1)
        v = v.unsqueeze(1)

        scores = torch.matmul(q, k.transpose(-2,-1)) * self.scale
        attn_weights = torch.softmax(scores,dim=-1)
        attn_weights = self.dropout(attn_weights)

        context = torch.matmul(attn_weights, v)
        context = context.transpose(1,2).view(B,T,self.input_dim)

        return self.out(self.dropout(context))


class GroupedMultiHeadAttention(nn.Module):
    '''
    - Each head has its own Q
    - Grouped K and Q
    '''

    def __int__(self, input_dim, num_heads, kv_heads, dropout=0.1) -> None:
        super().__init__()
        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"
        assert num_heads % kv_heads == 0, "input_dim must be divisible by num_heads"

        self.input_dim = input_dim
        self.num_heads = num_heads
        self.kv_heads = kv_heads
        self.group_size = num_heads // kv_heads
        self.head_dim = input_dim // num_heads
        self.scale = 1.0/math.sqrt(self.head_dim)

        self.q = nn.Linear(input_dim, input_dim, bias=False)
        self.kv = nn.Linear(input_dim, kv_heads* 2* self.head_dim, bias= False)
```

```python
        self.out = nn.Linear(input_dim, input_dim, bias=True)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, _ = x.shape

        q = self.q(x).view(B,T,self.num_heads,self.head_dim)
        q = q.transpose(1,2)

        # Key difference from MQA
        kv = self.kv(x).view(B,T, self.kv_heads, 2*self.head_dim).transpose(1,2)
        k, v = kv.chunk(2, dim=-1)
        k = k.repeat_interleave(self.group_size, dim=1)  # (B, H_q, T, D)
        v = v.repeat_interleave(self.group_size, dim=1)


        scores = torch.matmul(q, k.transpose(-2,-1)) * self.scale
        attn_weights = torch.softmax(scores,dim=-1)
        attn_weights = self.dropout(attn_weights)

        context = torch.matmul(attn_weights, v)
        context = context.transpose(1,2).view(B,T,self.input_dim)

        return self.out(self.dropout(context))
```