projects/project_3/code/model.py

```
## Building and training a bigram language model
import math


import torch
import torch.nn as nn

from config import BigramConfig

device = torch.device(
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.mps.is_available()
    else "cpu"
)


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

    def __init__(self, config: BigramConfig):
        """
        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.config = config
        self.embeddings = nn.Embedding(self.config.vocab_size, self.config.embed_dim)
        self.linear = nn.Linear(
            self.config.context_length * self.config.embed_dim,
            self.config.vocab_size,
            bias=True,
        )
        self.dropout = nn.Dropout(p=self.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 ========= #

        embed = self.embeddings(x).squeeze(1)  # (batch_size, embed_dim)
        out = self.linear(embed)  # (batch_size, vocab_size)
        out = self.dropout(out)  # (batch_size, vocab_size)

        return out

        # ========= 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):
            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 ========= ###

        f = torch.softmax
        context = torch.tensor(context, device=device)
        current_token = context[-1]
        for _ in range(max_new_tokens):
            logits = self.forward(
                torch.tensor([current_token], device=device)
            ).squeeze()
            probabilities = f(logits, dim=0)
            current_token = torch.multinomial(probabilities, 1).to(device)
            context = torch.cat((context, current_token), dim=0)

        return context
        ### ========= 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__(
        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(
            self.input_dim, self.output_key_query_dim, bias=False, device=device
        )
        self.query = nn.Linear(
            self.input_dim, self.output_key_query_dim, bias=False, device=device
        )
        self.value = nn.Linear(
            self.input_dim, self.output_value_dim, bias=False, device=device
        )

        self.dropout = nn.Dropout(p=dropout)

        causal_mask = torch.ones((max_len, max_len), device=device)
        causal_mask *= -float("inf")
        causal_mask = torch.triu(causal_mask, 1)

        # ========= 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
            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 ========= #

        _, num_tokens, _ = x.shape
        dk = self.output_key_query_dim
        K = self.key(x)  # (B, T, Dk)
        Q = self.query(x)  # (B, T, Dk)
        V = self.value(x)  # (B, T, Dv)

        attention_scores = Q @ K.transpose(-2, -1)  # (B, T, T)
        attention_scores /= math.sqrt(dk)  # (B, T, T)

        mask = self.causal_mask[:num_tokens, :num_tokens]  # (T, T)
        mask = mask.unsqueeze(0)  # not sure this is necessary # (1, T, T)
        mask = mask.type(torch.bool)

        mask = torch.where(mask, -torch.inf, 0)

        attention_scores += mask  # (B, T, T)

        attention_weights = torch.softmax(attention_scores, dim=-1)  # (B, T, T)
        attention_weights = self.dropout(attention_weights)  # (B, T, T)

        out = attention_weights @ V  # (B, T, Dv)
        return out
        # ========= 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
        self.head_dim = input_dim // num_heads

        self.heads = []
        for i in range(num_heads):
            head = SingleHeadAttention(
                input_dim=input_dim,
                output_key_query_dim=self.head_dim,
                output_value_dim=self.head_dim,
            )
            setattr(self, f"head_{i}", head)
            self.heads.append(head)

        self.out = nn.Linear(self.input_dim, self.input_dim)
        self.dropout = nn.Dropout(p=dropout)

        # ========= 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 ========= #

        head_outputs = []
        for head in self.heads:
            head_outputs.append(head(x))

        out = self.out(torch.cat(head_outputs, dim=-1))
        out = self.dropout(out)
        return out

        # ========= 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.
        """
        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)
        out = self.activation(out)
        out = self.fc2(out)
        out = self.dropout(out)
        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.
        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 ========= #

        mean = torch.mean(input, dim=2, keepdim=True)
        var = torch.var(input, dim=2, keepdim=True, correction=0)

        # ========= 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))
```

```python
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(normalized_shape=input_dim)
        self.attention = MultiHeadAttention(input_dim=input_dim, num_heads=num_heads)
        self.norm2 = LayerNorm(normalized_shape=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.
        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 ========= #

        intermediate = self.norm1(x)
        intermediate = self.attention(intermediate)
        intermediate += x
        out = self.norm2(intermediate)
        out = self.feedforward(out)
        out += intermediate

        return out

        # ========= 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.context_length = config.context_length
        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)
            ]
        )

        # 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 ========= ###

        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.transformer_layers:
            x = layer(x)

        # language modeling head
        x = self.prehead_norm(x)
        logits = self.head(x)

        return logits
```

```python
        ### ========= 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 ========= ###

        f = torch.softmax
        context = torch.tensor(context, device=device).unsqueeze(0)
        for i in range(max_new_tokens):
            current_context = context[:, -(self.context_length) :]
            logits = self(current_context)
            next_token_logits = logits[:, -1, :]
            next_token_probabilities = f(next_token_logits, dim=-1)
            current_token = torch.multinomial(
                input=next_token_probabilities, num_samples=1
            ).to(device)
            context = torch.cat((context, current_token), dim=1)

        return context
        ### ========= TODO : END ========= ###


class SingleHeadGeneralAttention(nn.Module):
    def __init__(
        self,
        input_dim,
        output_key_query_dim=None,
        output_value_dim=None,
        dropout=0.1,
        max_len=512,
    ):
        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

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

        self.dropout = nn.Dropout(p=dropout)

        causal_mask = torch.ones((max_len, max_len), device=device)
        causal_mask *= -float("inf")
        causal_mask = torch.triu(causal_mask, 1)

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

    def forward(self, x_query, x_key, x_value):
        _, num_tokens, _ = x_query.shape
        dk = self.output_key_query_dim
        K = self.key(x_key)  # (B, T, Dk)
        Q = self.query(x_query)  # (B, T, Dk)
        V = self.value(x_value)  # (B, T, Dv)

        attention_scores = Q @ K.transpose(-2, -1)  # (B, T, T)
        attention_scores /= math.sqrt(dk)  # (B, T, T)

        mask = self.causal_mask[:num_tokens, :num_tokens]  # (T, T)
        mask = mask.unsqueeze(0)
        mask = mask.type(torch.bool)
        mask = torch.where(mask, -torch.inf, 0)

        attention_scores += mask  # (B, T, T)

        attention_weights = torch.softmax(attention_scores, dim=-1)  # (B, T, T)
        attention_weights = self.dropout(attention_weights)  # (B, T, T)

        out = attention_weights @ V  # (B, T, Dv)
        return out


class MultiHeadGeneralAttention(nn.Module):
    def __init__(self, input_dim, num_heads, dropout=0.1) -> None:
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads
        self.head_dim = input_dim // num_heads

        self.heads = []
        for i in range(num_heads):
            head = SingleHeadGeneralAttention(
                input_dim=input_dim,
                output_key_query_dim=self.head_dim,
                output_value_dim=self.head_dim,
            )
            setattr(self, f"head_{i}", head)
            self.heads.append(head)

        self.out = nn.Linear(self.input_dim, self.input_dim)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, x_query, x_key, x_value):
        head_outputs = []
        for head in self.heads:
            head_outputs.append(head(x_query, x_key, x_value))

        out = self.out(torch.cat(head_outputs, dim=-1))
        out = self.dropout(out)
        return out


class GeneralTransformerLayer(nn.Module):
    def __init__(self, input_dim, num_heads, feedforward_dim=None):
        super().__init__()

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

    def forward(self, x):
        intermediate = self.norm1(x)
        intermediate = self.attention(intermediate)
        intermediate += x
        out = self.norm2(intermediate)
        out = self.feedforward(out)
        out += intermediate

        return out


class Encoder(nn.Module):
    def __init__(self, config) → None:
        super().__init__()

        self.context_length = config.context_length
        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)
            ]
        )

        # 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):
        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.transformer_layers:
            x = layer(x)

        return x


class Decoder(nn.Module):
    def __init__(self, config) -> None:
        super().__init__()

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

        # 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, encoder_output):
        B, T = x.shape

        # embeddings
        token_embeds = self.vocab_embedding(x)
        position_embeds = self.positional_embedding(self.pos[:T])
        position_embeds = position_embeds.unsqueeze(dim=0)
        x = token_embeds + position_embeds
        x = self.embed_dropout(x)

        # attention layers
        for layer in self.decoder_layers:
            x = layer(x_query=x, x_key=encoder_output, x_value=encoder_output)

        # language modeling head
        x = self.prehead_norm(x)
        logits = self.head(x)

        return logits

    def _init_weights(self, module):
        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):
        f = torch.softmax
        context = torch.tensor(context, device=device).unsqueeze(0)
        for i in range(max_new_tokens):
            current_context = context[:, -(self.context_length) :]
            logits = self(current_context)
            next_token_logits = logits[:, -1, :]
            next_token_probabilities = f(next_token_logits, dim=-1)
            current_token = torch.multinomial(
                input=next_token_probabilities, num_samples=1
            ).to(device)
            context = torch.cat((context, current_token), dim=1)

        return context
```