```
import torch


class BigramLanguageModel(nn.Module):
    # Constructor method for the BigramLanguageModel class
    def __init__(self):
        # Initialize the superclass (nn.Module) constructor
        super().__init__()
        # Initialize vocabulary, token embeddings, and other parameters and set to None
        self.vocab = None
        self.token_embeddings_table = None
        self.vocab_size = None
        self.encoder = None
        self.decoder = None
        # Initialize vocab size and set to int type
        self.vocab_size: int
        # Set the device to GPU if available, else CPU
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        # Initialize input length and batch size and set to None
        self.input_length = None
        self.batch_size = None

    # Forward method to define computation at every call
    def forward(self, in_ids, target=None):
        # Embed input ids using the token embeddings table
        in_ids_emb = self.token_embeddings_table(in_ids) # batch_size x vocab_size
        # Conditional block for calculating loss
        if target is None:
            ce_loss = None
        else:
            batch_size, input_length, vocab_size = in_ids_emb.shape
            token_rep = in_ids_emb.view(batch_size * input_length, vocab_size)
            targets = target.view(batch_size * input_length)
            # Calculate cross-entropy loss
            ce_loss = F.cross_entropy(token_rep, targets)
        return in_ids_emb, ce_loss

    # Fit method for training the model
    def fit(self, train_iters=100, eval_iters=10, lr=0.001):
        # Initialize Adam optimizer
        optimizer = torch.optim.Adam(self.parameters(), lr=lr)
        for iteration in range(train_iters):
            # Evaluate and print average loss at specified intervals
            if iteration % eval_iters == 0:
                avg_loss = self.eval_loss(eval_iters)
                print(f"iter {iteration} train {avg_loss['train']} val {avg_loss['eval']}")
            # Get batch of data and compute loss
            inputs, targets = self.get_batch(split='train')
            _, ce_loss = self(inputs, targets)
            # Clear gradients, backpropagate loss, and update model parameters
            optimizer.zero_grad(set_to_none=True)  # clear gradients of previous step
            ce_loss.backward()  # propagate loss back to each unit in the network
            optimizer.step()  # update network parameters w.r.t the loss

    # Generate method for text generation
    def generate(self, context_tokens, max_new_tokens):
        # Loop to generate the specified number of new tokens
        for _ in range(max_new_tokens):
            # Generate token representations for the current context tokens
            token_rep, _ = self(context_tokens)
            # Extract the representation of the last token in the sequence
            last_token_rep = token_rep[:, -1, :]
            # Compute the probabilities of the next token using softmax
            probs = F.softmax(last_token_rep, dim=-1)
            # Sample the next token based on the probabilities
            next_token = torch.multinomial(probs, 1)
            # Concatenate the next token to the context tokens
            context_tokens = torch.cat([context_tokens, next_token], dim=1)
            # Return the context tokens with the newly generated tokens
        return context_tokens

    # Preprocess text data
    def prep(self, text):
        # Split text into training and validation sets
        n = len(text)
        self.train_text = text[:int(n * 0.9)]
```

```
        self.val_text = text[int(n * 0.9):]

        # Convert text to tensors
        self.train_data = torch.tensor(self.encoder(self.train_text), dtype=torch.long)
        self.val_data = torch.tensor(self.encoder(self.val_text), dtype=torch.long)

        # Initialize token embeddings table
        self.token_embeddings_table = nn.Embedding(self.vocab_size, self.vocab_size)

    # Create batches of data
    def get_batch(self, split='train', input_length=8, batch_size=4):
        data = self.train_data if split == 'train' else self.val_data
        # Get random chunks of data
        ix = torch.randint(len(data) - input_length, (batch_size,))
        inputs_batch = torch.stack([data[i:i + input_length] for i in ix])
        targets_batch = torch.stack([data[i + 1:i + input_length + 1] for i in ix])
        # Move batches to the specified device
        inputs_batch = inputs_batch.to(self.device)
        targets_batch = targets_batch.to(self.device)
        return inputs_batch, targets_batch

# Sample text for training
text = 'a quick brown fox jumps over the lazy dog.\\n ' \
       'lazy dog and a quick brown fox.\\n' \
       'a dog is lazy and fox is quick.\\n' \
       'fox jumps and dog is lazy.\\n' \
       'dog is lazy and fox is brown.'

# Instantiate the BigramLanguageModel class
model = BigramLanguageModel()
# Move the model to the appropriate device (GPU or CPU)
model = model.to(model.device)
# Preprocess the text data for the model
model.prep(text)
# Filter the model parameters to only include those that require gradients
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
# Calculate and print the total number of trainable parameters in the model
print(f'params {sum([np.prod(p.size()) for p in model_parameters])}')
# Generate a batch of input and output data for training
input_batch, output_batch = model.get_batch(split='train')
# Pass the batch through the model (this step is likely for forward pass testing)
_, _ = model(input_batch, output_batch)
# Train the model with specified number of iterations, evaluation intervals, and learning rate
model.fit(train_iters=10000, eval_iters=500, lr=0.001)
# Generate text using the model, starting with an initial context of zeros
outputs = model.generate(context_tokens=torch.zeros((1, 1), dtype=torch.long,
                         device=model.device), max_new_tokens=100)
# Print the generated text outputs
print(outputs)
```