```
#Importing modules
import torch
from torch import nn
from torch.nn import functional as F


''' Look at the very last character to generate next
    @Author: Uzair Ahmad
    2022
'''


class BigramLanguageModel(nn.Module):    # Defines a class that inherits from nn.Module, making it a neural network module in PyTorch.
    def __init__(self):    # The __init__ method initializes the instance and declares several instance variables.
        super().__init__()
        self.vocab = None
        self.token_embeddings_table = None
        self.vocab_size = None
        self.encoder = None
        self.decoder = None
        self.vocab_size: int
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'   # Utilize a GPU if available, otherwise default to using the CPU.
        # input_length = how many consecutive tokens/chars in one input
        self.input_length = None
        # batch_size = how many inputs are going to be processed in-parallel (on GPU)
        self.batch_size = None

    def forward(self, in_ids, target=None):
        in_ids_emb = self.token_embeddings_table(in_ids) # uses an embedding table (`self.token_embeddings_table`) to convert input token IDs
        if target is None:    # The function checks if a target tensor is provided; if not, it skips computing the cross entropy loss.
            ce_loss = None
        else:
            # Reshapes the tensor for cross entropy loss calculation, provided a target tensor is available.
            batch_size, input_length, vocab_size = in_ids_emb.shape   # Retrieves the dimensions of the input embeddings.
            token_rep = in_ids_emb.view(batch_size * input_length, vocab_size)  # Reshapes the input embeddings to merge the batch and input
            targets = target.view(batch_size * input_length)  # `in_ids_emb` is being transformed from a three-dimensional tensor to a two-di
            ce_loss = F.cross_entropy(token_rep, targets)   # Calculates the loss between the predictions (`token_rep`) and the actual target
        return in_ids_emb, ce_loss

    def fit(self, train_iters=100, eval_iters=10, lr=None):  # Training the language model.
        learning_rate = lr if lr is not None else 0.01
        optimizer = torch.optim.Adam(self.parameters(), lr=learning_rate)  #  Initializes an Adam optimizer, to be used for updating the mode
        for iteration in range(train_iters):  # Each iteration signifies a single step in the training process.
            if iteration % (train_iters // 20) == 0:
                avg_loss = self.eval_loss(eval_iters)
                print(f"iter {iter} train {avg_loss['train']} val {avg_loss['eval']}")
            inputs, targets = self.get_batch(split='train')   # Retrieves a batch of input-target pairs from the training data, where the inp
            _, ce_loss = self(inputs, targets)
            optimizer.zero_grad(set_to_none=True)  # Clears gradients from the previous step.
            ce_loss.backward()  # Backpropagates the loss to each unit in the network.
            optimizer.step()  # Updates network parameters with respect to the computed loss.

    def generate(self, context_tokens, max_new_tokens):
        for _ in range(max_new_tokens):  # During each iteration, the model predicts the next token using the current `context_tokens`. The `
            token_rep, _ = self(context_tokens)
            last_token_rep = token_rep[:, -1, :]   # Retrieves the representation of the last token in the sequence.
            probs = F.softmax(last_token_rep, dim=1)  # Applies the softmax function to the representation of the last token, converting the 
            next_token = torch.multinomial(probs, num_samples=1)   # Samples a token from the probability distribution, introducing randomnes
            context_tokens = torch.cat((context_tokens, next_token), dim=1)  # Updates `context_tokens` by appending the newly predicted `nex
        output_text = self.decoder(context_tokens[0].tolist())   # Converts the sequence of token IDs back into human-readable text.
        return output_text

    @torch.no_grad()  # Instructs PyTorch to not prepare for back-propagation, typically using a context manager.
    def eval_loss(self, eval_iters):
        perf = {}
        # Before running inference, set dropout and batch normalization layers to evaluation mode.
        self.eval()
        for split in ['train', 'eval']:
            losses = torch.zeros(eval_iters)
            for k in range(eval_iters):
                tokens, targets = self.get_batch(split)  # Retrieves a random batch of inputs and targets.
                _, ce_loss = self(tokens, targets)  # Performs the forward pass, which computes the model's predictions.
                losses[k] = ce_loss.item()  # Obtains the value of the loss tensor as a standard Python number.
            perf[split] = losses.mean()
        self.train()  # trains the model reccursively
        return perf
    def prep(self, text):
        self.vocab = sorted(list(set(text)))  # Generates a vocabulary by identifying all unique characters in the input text, converting the
        self.vocab_size = len(self.vocab)
        ctoi = {c: i for i, c in  # Constructs a dictionary that associates each character (c) in the vocabulary with a unique integer (i). Th
                enumerate(self.vocab)}  # Establishes a mapping from characters (c) to integers (i), creating a character-to-integer (c-to-i)
        itoc = {i: c for c, i in ctoi.items()}  # Creates a mapping from integers (i) to characters (c), forming an integer-to-character (i-to

        self.encoder = lambda text: [ctoi[c] for c in text]  # Converts a sequence of characters into a sequence of corresponding indices usin
        self.decoder = lambda nums: ''.join([itoc[i] for i in nums])  #  Conducts the reverse operation, transforming a sequence of indices ba

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

        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)

        # The model will convert each input token into a vector of size `vocab_size`.
        self.token_embeddings_table = \
            nn.Embedding(self.vocab_size, self.vocab_size)

    def get_batch(self, split='train', input_length=8, batch_size=4): # Responsible for generating data batches for model training or evaluati
        data = self.train_data if split == 'train' else self.val_data
        # Retrieve random chunks of data with a length of `batch_size`.
        ix = torch.randint(len(data) - input_length, (batch_size,)) # Generates `batch_size` random starting points (`ix`) for sequences in th
        inputs_batch = torch.stack([data[i:i + input_length] for i in ix])  # It is generated by slicing the data from each starting index.
        targets_batch = torch.stack([data[i + 1:i + input_length + 1] for i in ix])  # It is created in a similar manner, with each sequence s
        inputs_batch = inputs_batch.to(self.device)   # Ensures that both the input and target batches are transferred to the appropriate devi
        targets_batch = targets_batch.to(self.device) # Ensures that both the input and target batches are transferred to the appropriate devi
        return inputs_batch, targets_batch


import numpy as np
# Sample text for training
text = 'a quick brown fox jumps of the lazy dog.\n ' \
       'a quick brown fox jumps of the lazy dog.\n' \
       'a quick brown fox jumps of the lazy dog.\n' \
       'a quick brown fox jumps of the lazy dog.\n' \
       'a quick brown fox jumps of the lazy dog.'

# 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)
```

```
params 784
iter <built-in function iter> train 3.7328264713287354 val 3.6603426933288574
iter <built-in function iter> train 3.1189944744110107 val 3.0882856845855713
iter <built-in function iter> train 2.590837240219116 val 2.565760850906372
iter <built-in function iter> train 2.1379659175872803 val 2.13832426071167
iter <built-in function iter> train 1.759172797203064 val 1.7684252262115479
iter <built-in function iter> train 1.4687942266464233 val 1.5164238214492798
iter <built-in function iter> train 1.2422937154769897 val 1.300957202911377
iter <built-in function iter> train 1.0767475366592407 val 1.1384706497192383
iter <built-in function iter> train 0.9663461446762085 val 1.0236057043075562
iter <built-in function iter> train 0.8843268752098083 val 0.9516713619232178
iter <built-in function iter> train 0.8184308409690857 val 0.9016019105911255
iter <built-in function iter> train 0.7866175174713135 val 0.8649009466171265
iter <built-in function iter> train 0.7601588368415833 val 0.8382517099380493
iter <built-in function iter> train 0.7384308576583862 val 0.8089299201965332
iter <built-in function iter> train 0.72303706407547 val 0.7939233779907227
iter <built-in function iter> train 0.7131791710853577 val 0.776553213596344
iter <built-in function iter> train 0.7023618221282959 val 0.7672250866889954
iter <built-in function iter> train 0.7039012908935547 val 0.7707991003990173
iter <built-in function iter> train 0.6891452670097351 val 0.768969714641571
iter <built-in function iter> train 0.6961244940757751 val 0.7667611241340637

 lazy dog.
a lazy qumps the dof f the by dox juick lazy own a the brown la ox qumps ox f dofof lazy
```