```python
"""
Training file for the models we implemented
"""

from pathlib import Path

import torch
import torch.nn as nn
import torch.nn.utils
import torch.optim as optim
from torch.utils.data import DataLoader
from einops import rearrange
import wandb

from model import BigramLanguageModel, MiniGPT
from dataset import TinyStoriesDataset
from config import BigramConfig, MiniGPTConfig

def solver(model_name):
    # Initialize the model
    if model_name == "bigram":
        config = BigramConfig
        model = BigramLanguageModel(config)
    elif model_name == "minigpt":
        config = MiniGPTConfig
        model = MiniGPT(config)
    else:
        raise ValueError("Invalid model name")

    # Load the dataset
    train_dataset = TinyStoriesDataset(
        config.path_to_data,
        mode="train",
        context_length=config.context_length,
    )
    eval_dataset = TinyStoriesDataset(
        config.path_to_data, mode="test", context_length=config.context_length
    )

    # Create the dataloaders
    train_dataloader = DataLoader(
        train_dataset, batch_size=config.batch_size, pin_memory=True
    )
    eval_dataloader = DataLoader(
        eval_dataset, batch_size=config.batch_size, pin_memory=True
    )

    # Set the device
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Print number of parameters in the model
    def count_parameters(model):
        return sum(p.numel() for p in model.parameters() if p.requires_grad)
    print("number of trainable parameters: %.2fM" % (count_parameters(model) / 1e6,))

    # Initialize wandb if you want to use it
    if config.to_log:
        wandb.init(project="dl2_proj3")

    print("Eval Logging Inteval:" , config.log_interval )

    # Create the save path if it does not exist
    if not Path.exists(config.save_path):
        Path.mkdir(config.save_path, parents=True, exist_ok=True)

    ### ==================== START OF YOUR CODE ==================== ###
    """
```

```python
    You are required to implement the training loop for the model.

    The code below is a skeleton for the training loop, for your reference.
    You can fill in the missing parts or completely set it up from scratch.

    Please keep the following in mind:
    - You will need to define an appropriate loss function for the model.
    - You will need to define an optimizer for the model.
    - You are required to log the loss (either on wandb or any other logger you prefer) every
`config.log_interval` iterations.
    - It is recommended that you save the model weights every `config.save_iterations`
iterations. You can also just save the model with the best training loss.

    NOTE :
    - Please check the config file to see the different configurations you can set for the
model.
    - The MiniGPT config has params that you do not need to use, these were added to scale the
model but are
    not a required part of the assignment.
    - Feel free to experiment with the parameters and I would be happy to talk to you about
them if interested.
    """

    ### ========= TODO : START ========= ###
    # Define the loss function
    lossf = nn.CrossEntropyLoss()

    # Define the optimizer
    optimizer =  optim.Adam(model.parameters(), lr=0.0001)
    ### ======== TODO : END ========= ###

    if config.scheduler:
        scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
            optimizer, T_0=2000, T_mult=2
        )

    model.train()
    model.to(device)

    best_eval_loss = float("inf")
    eval_patience = 3   # how many times to tolerate no improvement
    eval_no_improve_count = 0
    print("Total number of training set: ", len(train_dataloader))
    print("Total number of eval iterations: ", len(eval_dataloader))
    for i, (context, target) in enumerate(train_dataloader):


        context= context.to(device)
        target = target.to(device)

        train_loss = 0.0 # You can use this variable to store the training loss for the current
iteration
        ### ======== TODO : START ========= ###
        # Do the forward pass, compute the loss, do the backward pass, and update the weights
with the optimizer.
        model.zero_grad()
        logits = model(context)
        B, T, V = logits.shape
        # print(B,T,V)
        logits = logits.view(B * T, V)
        target = target.view(-1)
        loss = lossf(logits, target)

        loss.backward()
        optimizer.step()

        # Gather data and report
```

```
        train_loss += loss.item()
        if i % 1000 == 999:
            last_loss = train_loss / 1000 # loss per batch
            print('  batch {} loss: {}'.format(i + 1, last_loss))
            # tb_x = epoch_index * len(training_loader) + i + 1
            # tb_writer.add_scalar('Loss/train', last_loss, tb_x)
            running_loss = 0.

        if i >= len(train_dataloader): # config.batch_size:
            print("Loop Exceeding Number of batches")
            break


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

        if config.scheduler:
            scheduler.step()

        del context, target # Clear memory


        # print(torch.cuda.memory_summary())
        if i % config.log_interval == 0:
            # print("Evaluating Model", i)
            model.eval()
            eval_loss = 0.0 # You can use this variable to store the evaluation loss for the
current iteration
            total_samples = 0
            ### ======== TODO : START ========= ###
            # Compute the evaluation loss on the eval dataset.

            with torch.no_grad():
                for val_context, val_target in eval_dataloader:
                    val_context = val_context.to(device)
                    val_target = val_target.to(device)

                    val_logits = model(val_context)
                    B, T, V = val_logits.shape
                    val_logits = val_logits.view(B * T, V)
                    val_target = val_target.view(-1)

                    val_loss = lossf(val_logits, val_target)
                    eval_loss += val_loss.item() * B  # accumulate weighted by batch size
                    total_samples += B

                    eval_loss_temp = eval_loss / total_samples  # average over dataset
                    # Early stopping
                    if eval_loss_temp < best_eval_loss:
                        best_eval_loss = eval_loss_temp
                        eval_no_improve_count = 0
                    else:
                        eval_no_improve_count += 1
                        # print(f"No improvement in eval loss. Count =
{eval_no_improve_count}/{eval_patience}")

                    if eval_no_improve_count >= eval_patience:
                        # print("Early stopping triggered.")
                        break
            eval_loss /= total_samples  # average over dataset

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

            # print(
            #     f"Iteration {i}, Train Loss: {train_loss}",
            #     f"Eval Loss: {eval_loss}",
            # )
```

```
            if config.to_log:
                wandb.log(
                    {
                        "Train Loss": train_loss,
                        "Eval Loss": eval_loss,
                    }
                )

            model.train()



        # Save the model every config.save_iterations
        if i % config.save_iterations == 0:
            torch.save(
                {
                    "model_state_dict": model.state_dict(),
                    "optimizer_state_dict": optimizer.state_dict(),
                    "train_loss": train_loss,
                    "eval_loss": eval_loss,
                    "iteration": i,
                },
                config.save_path / f"mini_model_checkpoint_{i}.pt",
            )

        if i > config.max_iter:
            break
```