projects/project_3/code/train.py

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

from pathlib import Path

import torch
import torch.nn.utils
import torch.nn as nn
from torch.utils.data import DataLoader
import wandb
from tqdm import tqdm

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(to_log=True, log_interval=10_000)
        model = BigramLanguageModel(config)
    elif model_name == "minigpt":
        config = MiniGPTConfig(
            to_log=True, save_iterations=100_000, log_interval=100_000
        )
        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 "mps"
        if torch.mps.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.login()
        wandb.init(
            project="dl2_proj3",
            config={
                "learning_rate": config.learning_rate,
                "architecture": model_name,
                "dataset": "TinyStories",
            },
        )

    # 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 ==================== ###
    """
    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
    loss = nn.CrossEntropyLoss()

    # Define the optimizer
    optimizer = torch.optim.Adam(params=model.parameters(), lr=config.learning_rate)
    ### ======== TODO : END ========= ###

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

    model.train()
    model.to(device)

    max_iter_eval = len(eval_dataloader) // 500

    for i, (context, target) in tqdm(enumerate(train_dataloader)):
        context = context.to(device)
        target = target.to(device)
        train_loss = 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)

        target = target.long()

        if model_name == "bigram":
            target = target.squeeze()

        elif model_name == "minigpt":
            B, T, _ = logits.shape
            logits = logits.reshape(B * T, -1)
            target = target.reshape(B * T)

        batch_loss = loss(logits, target)

        batch_loss.backward()
        optimizer.step()

        train_loss += batch_loss.item()
```

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

        if config.scheduler:
            scheduler.step()

        del context, target  # Clear memory

        if i % config.log_interval == 0:
            model.eval()
            eval_loss = 0  # You can use this variable to store the evaluation loss for
the current iteration

            ### ======== TODO : START ========= ###
            # Compute the evaluation loss on the eval dataset.

            with torch.no_grad():
                for j, (context, target) in enumerate(eval_dataloader):
                    context = context.to(device)
                    target = target.to(device)

                    target = target.long()
                    logits = model(context)

                    if model_name == "bigram":
                        target = target.squeeze()

                    elif model_name == "minigpt":
                        logits = logits.reshape(B * T, -1)
                        target = target.reshape(B * T)

                    batch_loss = loss(logits, target)
                    eval_loss += batch_loss.item()
                    del context, target  # Clear memory

                    if j > max_iter_eval:
                        break

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

            eval_loss /= max_iter_eval
            print(
                f"Iteration {i}\nTrain Loss: {train_loss}\nAverage 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
```