
Introduction

This project focuses on building a Large Language Model (LLM) from scratch using a transformer-based architecture. The goal is to provide an educational framework for understanding how modern NLP models are constructed, trained, and fine-tuned. The repository includes the necessary scripts for preprocessing data, defining the model architecture, and training the model on a custom dataset. By following this guide, developers and researchers can gain hands-on experience in the process of creating a state-of-the-art language model.

Prerequisites

To successfully build and train the Large Language Model (LLM) from scratch, ensure your environment meets the following prerequisites:

1. Python Version

- **Python 3.7 or higher:** The project is compatible with Python 3.7 and above.

2. Dependencies

Install the necessary Python libraries using `pip`. The primary dependencies include:

- **PyTorch:** A deep learning framework essential for model development and training.
- **Transformers:** A library by Hugging Face that provides pre-built transformer models and tokenizers.
- **NumPy:** A fundamental package for numerical computations.
- **Pandas:** A data manipulation and analysis library.
- **Matplotlib:** A plotting library for creating static, animated, and interactive visualizations.
- **TensorBoard:** A tool for visualizing training metrics.
- **TQDM:** A library for progress bars in loops.
- **Scikit-learn:** A machine learning library for data mining and data analysis.
- **Requests:** A simple HTTP library for Python.
- **OpenWebText:** A dataset used for training the model.

To install these dependencies, you can use the following command:

```
pip install torch transformers numpy pandas matplotlib tensorboard tqdm scikit-learn requests openwebtext
```

3. Hardware Requirements

- **GPU Support:** Training large models requires significant computational power. It's recommended to use GPUs (e.g., NVIDIA Tesla V100, A100) to accelerate the training process.
- **Memory:** Ensure your system has sufficient RAM and GPU memory to handle large datasets and model parameters.

4. Environment Setup

- **Virtual Environment:** It's advisable to create a virtual environment to manage dependencies and avoid conflicts. You can set up a virtual environment using `venv` or `conda`:

Using `venv`:

```
python -m venv llm_env
```

```
source llm_env/bin/activate # On Windows, use llm_env\Scripts\activate
```

○

Using `conda`:

```
conda create --name llm_env python=3.7
```

```
conda activate llm_env
```

○

- **CUDA:** If using NVIDIA GPUs, ensure that the appropriate CUDA version is installed to enable GPU acceleration.

5. Dataset

- **OpenWebText:** The model is trained on the OpenWebText dataset, which is a collection of web pages extracted from Reddit submissions. You can download and preprocess the dataset using the provided scripts in the repository.

File Structure

The project is organized into several key directories and files, each serving a specific purpose. Below is an overview of the main components of the repository (excluding the `Resources` folder):

1. **data/**

This folder contains the datasets required for training the model. It may include:

- Preprocessed text data
- Scripts for downloading and cleaning datasets (e.g., OpenWebText)

2. **notebooks/**

Jupyter notebooks for experimentation and model evaluation. Notebooks might include:

- Exploratory Data Analysis (EDA)
- Model training and evaluation
- Hyperparameter tuning experiments

3. **src/**

The source code for the project, containing the core implementation:

- **model.py**: Defines the architecture of the transformer-based language model.
- **train.py**: Script for training the model, including data loading, model initialization, and training loop.
- **utils.py**: Utility functions for data processing, tokenization, and other auxiliary tasks.
- **config.py**: Configuration file for model parameters, training settings, and hyperparameters.

4. **scripts/**

This directory includes various helper scripts for:

- Data preprocessing
- Model evaluation
- Experimentation and analysis

5. **requirements.txt**

A file that lists all the dependencies required to run the project. This file can be used to install the necessary libraries using **pip**:

```
pip install -r requirements.txt
```

6. **README.md**

The main documentation file for the project. It includes:

- Project overview
- Setup instructions
- Usage examples

7. LICENSE

The license under which the project is distributed, detailing the terms of use and redistribution.

Core Components

The project consists of several key components that together form the core functionality of building and training a large language model. Below is an overview of the primary components:

1. Model Architecture (`model.py`)

This file defines the architecture of the transformer-based language model. It includes:

- **Model Structure:** The design of the neural network, encompassing layers such as attention mechanisms, feedforward networks, and normalization layers.
- **Configuration Parameters:** Specifications for the number of layers, hidden units, attention heads, and other architectural details.
- **Forward Pass Implementation:** The method that outlines the data flow through the model during both training and inference phases.

2. Training Script (`train.py`)

The `train.py` script is responsible for training the model. It includes:

- **Data Loading:** Functions to load and preprocess the dataset, ensuring it's in the appropriate format for training.
- **Model Initialization:** Code to instantiate the model with the specified configuration parameters.
- **Training Loop:** The iterative process where the model learns from the data, including loss computation, backpropagation, and weight updates.
- **Evaluation:** Periodic assessment of the model's performance on validation data to monitor progress and prevent overfitting.

3. Utility Functions (`utils.py`)

This file contains utility functions used across the project, including:

- **Data Preprocessing:** Functions for cleaning and tokenizing the text data, preparing it for model input.
- **Metric Calculation:** Functions to compute performance metrics such as accuracy, perplexity, and loss.
- **Model Checkpoints:** Functions to save and load model weights, enabling the continuation of training or inference at a later time.

4. Configuration (`config.py`)

The `config.py` file contains the configuration settings for the model, including:

- **Hyperparameters:** Learning rate, batch size, number of epochs, and other training-related parameters.
- **Model Parameters:** Details such as the number of layers, hidden units, attention heads, and other architectural specifications.
- **File Paths:** Directories for saving models, logs, and other outputs.

5. Data Preprocessing Scripts (**scripts/**)

The **scripts/** directory contains scripts for data preprocessing, including:

- **Dataset Downloading:** Scripts to download datasets like OpenWebText.
- **Data Cleaning:** Functions to remove noise and irrelevant information from the text data.
- **Tokenization:** Scripts to convert text into tokens, preparing it for model input.
- **Data Splitting:** Functions to divide the dataset into training, validation, and test sets.

6. Evaluation and Experimentation Notebooks (**notebooks/**)

The **notebooks/** directory includes Jupyter notebooks for:

- **Exploratory Data Analysis (EDA):** Visualizations and analyses to understand the dataset's characteristics.
- **Model Training and Evaluation:** Notebooks to train the model and evaluate its performance on various tasks.
- **Hyperparameter Tuning:** Experiments to find the optimal hyperparameters for the model.
- **Visualization:** Plots and graphs to visualize training progress, loss curves, and other metrics.

Text Data Source: "The Wonderful Wizard of Oz"

It appears that the **wizard_of_oz.txt** file in your repository contains the full text of "The Wonderful Wizard of Oz" by L. Frank Baum. This classic novel is available for free online through various sources, including Project Gutenberg.

If your project involves processing this text for training a language model, the training process would typically involve the following steps:

1. Data Collection

Gather the text data, which in this case is the content of "The Wonderful Wizard of Oz."

2. Data Preprocessing

- **Cleaning:** Remove any unwanted characters, formatting, or metadata.
- **Tokenization:** Convert the text into tokens (words or subwords) that the model can process.

- **Encoding:** Map tokens to numerical representations using techniques like one-hot encoding or embeddings.

3. Model Architecture

Define the structure of the language model, such as a transformer-based architecture.

4. Training

- **Input Preparation:** Organize the data into sequences suitable for training.
- **Optimization:** Use algorithms like stochastic gradient descent to minimize the loss function.
- **Evaluation:** Assess the model's performance on validation data to monitor for overfitting.

5. Fine-Tuning

Adjust hyperparameters and model configurations based on evaluation results to improve performance.

6. Deployment

Once trained, the model can generate text, answer questions, or perform other language-related tasks.

PART-1 :

1. Library Imports

```
import torch
```

```
import torch.nn as nn
```

```
from torch.nn import functional as F
```

```
import mmap
```

```
import random
```

```
import pickle
```

```
import argparse
```

- **torch:** This is the core library for deep learning using PyTorch, providing tensors, gradients, and GPU acceleration.

- **torch.nn**: Contains classes and functions to help build neural network layers.
 - **torch.nn.functional**: Provides functions for operations such as activation functions, loss functions, etc.
 - **mmap**: A memory-mapped file object to manage large datasets that cannot fit into memory directly.
 - **random**: Used for generating random numbers, possibly for data augmentation or shuffling.
 - **pickle**: A module for serializing and deserializing Python objects, potentially for saving models or datasets.
 - **argparse**: A standard Python library for parsing command-line arguments.
-

2. Command-line Argument Setup (Commented Out)

```
# parser = argparse.ArgumentParser(description='This is a demonstration program')

# Here we add an argument to the parser, specifying the expected type, a help message, etc.

# parser.add_argument('-batch_size', type=str, required=True, help='Please provide a
batch_size')

# args = parser.parse_args()

# Now we can use the argument value in our program.

# print(f'batch size: {args.batch_size}')
```

- **argparse.ArgumentParser**: Used to define expected command-line arguments and their properties.
- **add_argument**: Specifies that the argument `-batch_size` is required and should be a string. The `help` parameter gives a description of the argument.
- **parse_args()**: Parses the command-line arguments when the script is executed, making them accessible in `args`.

The code is commented out, meaning it won't run unless the comments are removed.

3. Device Selection: GPU or CPU

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

- This line checks if CUDA (GPU support) is available. If it is, the model will run on the GPU (**cuda**), otherwise, it will default to the CPU. This is important for optimizing performance, especially when training large models.

4. Hyperparameters Setup

```
batch_size = 32
```

```
block_size = 128
```

```
max_iters = 3000
```

```
learning_rate = 3e-4
```

```
eval_iters = 50
```

```
n_embd = 384
```

```
n_head = 4
```

```
n_layer = 4
```

```
dropout = 0.2
```

- **batch_size**: Number of samples processed in one forward pass through the model.
- **block_size**: This could represent the sequence length or block size of input data the model will process.
- **max_iters**: Maximum number of iterations or training steps to perform.
- **learning_rate**: The rate at which the model adjusts the weights during training. A small value prevents the model from changing too drastically.
- **eval_iters**: Number of iterations to run during evaluation phases (often used to compute validation loss).
- **n_embd**: The size of the embedding layer, typically related to the size of the model's input representation.

- **n_head**: Number of attention heads in multi-head attention, commonly used in transformer-based models.
 - **n_layer**: Number of layers in the model (e.g., transformer layers).
 - **dropout**: Dropout rate to prevent overfitting by randomly setting a fraction of input units to zero during training.
-

5. Print Device

```
print(device)
```

- This line prints out whether the model will run on the CPU or GPU (**cuda**) based on the earlier check.
-

Example

Let's simulate a situation where we run this script on a machine with a GPU available. If you use command-line arguments, you would typically pass the **batch_size** as follows:

```
python script_name.py -batch_size 64
```

This command would execute the script with a batch size of 64. Since the **argparse** section is commented out, it will use the default **batch_size** of 32.

In this case, the script will print:

```
cuda
```

This indicates that the model will use the GPU for training. If CUDA is not available, it would print **cpu**, meaning it will fall back on the CPU.

PART-2

1. Opening and Reading the Text File

with open('wizard_of_oz.txt', 'r', encoding='utf-8') as f:

```
text = f.read()
```

- **open('wizard_of_oz.txt', 'r', encoding='utf-8')**: Opens the file `wizard_of_oz.txt` in read mode (`'r'`) with UTF-8 encoding. The file is read as a text file.
 - **with**: A context manager that ensures the file is properly closed after reading, even if an error occurs.
 - **f.read()**: Reads the entire content of the file into the variable `text`.
-

2. Extracting Unique Characters

```
chars = sorted(set(text))
```

- **set(text)**: Converts the text into a set, which removes any duplicate characters and leaves only unique characters.
 - **sorted()**: Sorts the unique characters in ascending order (alphabetical order for characters).
 - **chars**: A sorted list of unique characters found in the file.
-

3. Printing the Unique Characters

```
print(chars)
```

- This prints the sorted list of unique characters found in the `wizard_of_oz.txt` file.
-

4. Calculating Vocabulary Size

```
vocab_size = len(chars)
```

- **len(chars)**: Calculates the number of unique characters in the text.

- **vocab_size**: Stores the total number of unique characters (the "vocabulary size").
-

Example

Suppose the `wizard_of_oz.txt` file contains the following text:

The road to the Emerald City is paved with yellow bricks.

- The `set(text)` will convert it into unique characters (e.g., 'T', 'h', 'e', 'r', 'o', 'a', 'd', ' ', 't', 'm', 'l', 'c', 'y', 'i', 'z').

After sorting, the output will look like:

```
[' ', 'T', 'a', 'c', 'd', 'e', 'h', 'i', 'l', 'm', 'o', 'r', 't', 'y', 'z']
```

-
- The vocabulary size (**vocab_size**) will be **15** because there are 15 unique characters in the text.

PART-3

1. Creating the **string_to_int** Dictionary

```
string_to_int = { ch:i for i,ch in enumerate(chars) }
```

- **enumerate(chars)**: Enumerates over the `chars` list, yielding pairs of index (**i**) and character (**ch**).
 - **{ch: i for i, ch in enumerate(chars)}**: Creates a dictionary where the keys are characters (**ch**) and the values are their corresponding indices (**i**) in the sorted list of unique characters. This allows converting a character into its integer index.
 - **string_to_int**: A dictionary mapping each unique character to a unique integer.
-

2. Creating the **int_to_string** Dictionary

```
int_to_string = { i:ch for i,ch in enumerate(chars) }
```

- **enumerate(chars)**: Again enumerates over the **chars** list, yielding pairs of index (**i**) and character (**ch**).
 - **{i: ch for i, ch in enumerate(chars)}**: Creates a dictionary where the keys are integers (**i**) and the values are characters (**ch**). This allows converting an integer index back to its corresponding character.
 - **int_to_string**: A dictionary mapping each integer index to its corresponding character.
-

3. Creating the **encode** Function

```
encode = lambda s: [string_to_int[c] for c in s]
```

- **lambda s**: Defines an anonymous function that takes a string **s** as input.
 - **[string_to_int[c] for c in s]**: Iterates through each character **c** in the string **s** and looks up its corresponding integer index in the **string_to_int** dictionary.
 - **encode**: A function that converts a string **s** into a list of integers, where each integer represents a character in the string.
-

4. Creating the **decode** Function

```
decode = lambda l: ''.join([int_to_string[i] for i in l])
```

- **lambda l**: Defines an anonymous function that takes a list **l** (a list of integers) as input.
 - **[int_to_string[i] for i in l]**: Iterates through each integer **i** in the list **l** and looks up its corresponding character in the **int_to_string** dictionary.
 - **''.join()**: Joins the list of characters into a single string.
 - **decode**: A function that converts a list of integers back into the original string by looking up each integer in the **int_to_string** dictionary and concatenating the characters.
-

Example

Suppose the list **chars** contains the following unique sorted characters:

```
chars = [' ', 'T', 'a', 'c', 'd', 'e', 'h', 'i', 'l', 'm', 'o', 'r', 't', 'y', 'z']
```

string_to_int would look like:

```
{ ' ': 0, 'T': 1, 'a': 2, 'c': 3, 'd': 4, 'e': 5, 'h': 6, 'i': 7, 'l': 8, 'm': 9, 'o': 10, 'r': 11, 't': 12, 'y': 13, 'z': 14 }
```

-

int_to_string would look like:

```
{ 0: ' ', 1: 'T', 2: 'a', 3: 'c', 4: 'd', 5: 'e', 6: 'h', 7: 'i', 8: 'l', 9: 'm', 10: 'o', 11: 'r', 12: 't', 13: 'y', 14: 'z' }
```

-

For the string:

```
s = "The road"
```

encode(s) would return:

```
[1, 5, 2, 4, 0, 10, 3, 8]
```

-

For the list:

```
l = [1, 5, 2, 4, 0, 10, 3, 8]
```

decode(l) would return:

```
'The road'
```

-

These functions can be used to encode a string into a sequence of integers (useful for feeding into machine learning models) and decode it back into the original string.

PART-4

1. Converting the Encoded Text into a Tensor

```
data = torch.tensor(encode(text), dtype=torch.long)
```

- **encode(text)**: The `text` variable (which is the entire content of `wizard_of_oz.txt`) is encoded into a list of integers using the previously defined `encode` function. Each integer represents a character from the text.
 - **torch.tensor(..., dtype=torch.long)**: Converts the list of integers into a PyTorch tensor of type `torch.long` (which is the standard integer type for PyTorch).
-

2. Splitting the Data into Training and Validation Sets

```
n = int(0.8*len(data))
```

```
train_data = data[:n]
```

```
val_data = data[n:]
```

- **n = int(0.8 * len(data))**: Computes 80% of the total length of the `data` tensor (used for training).
 - **train_data = data[:n]**: Selects the first 80% of the data as the training set.
 - **val_data = data[n:]**: Selects the remaining 20% of the data as the validation set.
-

3. Defining the `get_batch` Function

```
def get_batch(split):
```

```
    data = train_data if split == 'train' else val_data
```

```
    ix = torch.randint(len(data) - block_size, (batch_size,))
```

```
    x = torch.stack([data[i:i+block_size] for i in ix])
```

```
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
```

```
    x, y = x.to(device), y.to(device)
```

```
    return x, y
```

- **split**: A string argument that specifies whether to fetch a batch from the training data (`'train'`) or the validation data (`'val'`).

- `data = train_data if split == 'train' else val_data`: Based on the value of `split`, the function selects either the `train_data` or `val_data`.
 - `ix = torch.randint(len(data) - block_size, (batch_size,))`: Generates `batch_size` random indices (`ix`) from the dataset. Each index corresponds to the starting position of a sequence (of length `block_size`).
 - `x = torch.stack([data[i:i+block_size] for i in ix])`: Creates a batch of input sequences `x` by slicing `data` from each random index `i` and taking the next `block_size` characters.
 - `y = torch.stack([data[i+1:i+block_size+1] for i in ix])`: Creates the target batch `y`, which is just the input sequence `x` shifted by one character (this is often used in language models where the target is the next character).
 - `x, y = x.to(device), y.to(device)`: Moves both the input `x` and the target `y` tensors to the appropriate device (`'cuda'` or `'cpu'`) for computation.
 - `return x, y`: Returns the input sequence `x` and the target sequence `y`.
-

Example

Let's assume the text file contains a small sample text, and the parameters are set as follows:

- `block_size = 4`
- `batch_size = 2`

If the encoded text (from the `encode` function) is:

```
data = torch.tensor([1, 5, 2, 4, 0, 10, 3, 8, 6, 7])
```

- **Training Data (`train_data`)**: Will contain the first 80% of the data, and Validation Data (`val_data`) will contain the remaining 20%.

For the `get_batch('train')` function:

1. **Random Indices (`ix`)**: If the random indices generated are `[0, 3]`:
 - **Input Sequences (`x`)**:
 - For index 0, the sequence would be `data[0:4] = [1, 5, 2, 4]`
 - For index 3, the sequence would be `data[3:7] = [4, 0, 10, 3]`
 - Thus, `x = [[1, 5, 2, 4], [4, 0, 10, 3]]`
 - **Target Sequences (`y`)**:

- For index 0, the shifted sequence would be `data[1:5] = [5, 2, 4, 0]`
- For index 3, the shifted sequence would be `data[4:8] = [0, 10, 3, 8]`
- Thus, `y = [[5, 2, 4, 0], [0, 10, 3, 8]]`

This approach is commonly used when preparing datasets for training language models, where the task is to predict the next character in a sequence.

PART-5

1. Decorator `@torch.no_grad()`

`@torch.no_grad()`

- `@torch.no_grad()`: This decorator disables gradient computation within the scope of the `estimate_loss()` function. This is useful for evaluation, as it reduces memory usage and computation time by not tracking gradients (which are only necessary for training).
- This is typically used when performing inference or evaluation, where gradients are not required.

2. Function Definition `estimate_loss()`

```
def estimate_loss():
```

```
    out = {}
```

```
    model.eval()
```

- `def estimate_loss()`: Defines the function `estimate_loss()` that will be used to estimate the average loss for both the training and validation datasets.
- `out = {}`: Initializes an empty dictionary `out` that will store the average losses for each dataset split ('train' and 'val').
- `model.eval()`: Switches the model into evaluation mode. In PyTorch, `eval()` is used to turn off dropout and batch normalization layers, which behave differently during training and evaluation.

3. Iterating Over Training and Validation Sets

for split in ['train', 'val']:

- **for split in ['train', 'val']:** Loops over both the training set ('train') and validation set ('val') to compute the average loss for each.

4. Initialising Losses and Looping Through Iterations

losses = torch.zeros(eval_iters)

for k in range(eval_iters):

 X, Y = get_batch(split)

 logits, loss = model(X, Y)

 losses[k] = loss.item()

- **losses = torch.zeros(eval_iters):** Initializes a tensor **losses** of zeros, which will store the individual losses for each evaluation iteration.
- **for k in range(eval_iters):** Iterates over the number of evaluation iterations specified by **eval_iters**.
 - **X, Y = get_batch(split):** Retrieves a batch of input data (X) and target data (Y) from the appropriate dataset ('train' or 'val') using the **get_batch()** function.
 - **logits, loss = model(X, Y):** Passes the input batch X and target batch Y through the model to get the predicted outputs (**logits**) and the corresponding loss.
 - **losses[k] = loss.item():** Saves the loss for this iteration in the **losses** tensor. **.item()** is used to extract the loss as a Python number (from a tensor).

5. Calculating and Storing Mean Loss

out[split] = losses.mean()

- `out[split] = losses.mean()`: Computes the mean of the losses collected over the `eval_iters` iterations for the current split (either 'train' or 'val') and stores the result in the `out` dictionary under the key corresponding to the split ('train' or 'val').
-

6. Returning Model to Training Mode

`model.train()`

- `model.train()`: Switches the model back into training mode, which re-enables dropout and batch normalization layers, if applicable.
-

7. Returning the Losses

`return out`

- `return out`: Returns the dictionary `out` that contains the average loss for both the training and validation splits.
-

Example Workflow

1. `model.eval()`: The model is set to evaluation mode to avoid updating internal parameters like weights during inference.
2. **Loss Estimation**: For both training and validation data splits, the function computes the loss over multiple iterations (`eval_iters`), collecting individual loss values and averaging them.
3. **Model Back to Training Mode**: After evaluation, the model is set back to training mode with `model.train()` to resume training.

Example Flow

- Suppose the `eval_iters` is set to 50, meaning the evaluation function will run 50 iterations for each of the 'train' and 'val' datasets.

- For each iteration, the model will receive a batch of data (X, Y) and compute the loss. After 50 iterations, the function computes the average of those losses for both training and validation sets.

The resulting `out` dictionary will look like:

```
out = {

    'train': average_train_loss,

    'val': average_val_loss

}
```

•

This function is useful for tracking how well the model is performing during evaluation and preventing overfitting by comparing the loss on both training and validation sets.

PART-6

1. Head Class: One Head of Self-Attention

```
class Head(nn.Module):
```

```
    """ one head of self-attention """
```

- The `Head` class defines one attention head in the self-attention mechanism, a fundamental part of transformer models. Each head is responsible for computing attention weights and applying them to input data.

2. Initialization (`__init__`) Method for `Head`

```
def __init__(self, head_size):
```

```
    super().__init__()
```

```
    self.key = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.query = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.value = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
```

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

- **key, query, value**: These are learned linear projections that map the input embeddings (**n_embd**) to three separate vectors (**head_size**) for the key, query, and value in the attention mechanism. These are crucial components for calculating attention scores.
 - **self.register_buffer('tril', ...)**: Creates a triangular matrix used for masking future positions in the self-attention mechanism. This is essential in autoregressive models like GPT, where the model is not allowed to "see" future tokens.
 - **self.dropout**: A dropout layer that randomly zeroes out some of the attention weights during training to help regularize the model and prevent overfitting.
-

3. Forward Pass for Head Class

```
def forward(self, x):
```

```
    B, T, C = x.shape
```

```
    k = self.key(x) # (B, T, hs)
```

```
    q = self.query(x) # (B, T, hs)
```

```
    wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, T)
```

```
    wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
```

```
    wei = F.softmax(wei, dim=-1) # (B, T, T)
```

```
    wei = self.dropout(wei)
```

```
    v = self.value(x) # (B, T, hs)
```

```
    out = wei @ v # (B, T, hs)
```

```
    return out
```

- **B, T, C = x.shape**: Extracts the batch size (**B**), sequence length (**T**), and embedding size (**C**) from the input tensor **x**.
- **k, q, v**: Computes the key, query, and value matrices by applying the respective linear layers.

- `wei = q @ k.transpose(-2,-1)`: Computes the raw attention weights using the dot product of queries and transposed keys. These weights indicate how much attention the model should give to each token in the sequence relative to others.
 - `wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))`: Applies the triangular mask (`tril`) to prevent attending to future tokens during training (this is crucial for autoregressive models).
 - `wei = F.softmax(wei, dim=-1)`: Normalizes the attention weights using the softmax function to get probabilities.
 - `wei = self.dropout(wei)`: Applies dropout to the attention weights for regularization.
 - `out = wei @ v`: Performs the weighted sum of the values, producing the final output for this attention head.
-

4. `MultiHeadAttention` Class: Multiple Heads of Self-Attention

`class MultiHeadAttention(nn.Module):`

`""" multiple heads of self-attention in parallel """`

- The `MultiHeadAttention` class defines a multi-head self-attention mechanism, where multiple attention heads operate in parallel. Each head focuses on different parts of the sequence to capture various dependencies.
-

5. Initialization (`__init__`) Method for `MultiHeadAttention`

`def __init__(self, num_heads, head_size):`

`super().__init__()`

`self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])`

`self.proj = nn.Linear(head_size * num_heads, n_embd)`

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

- `self.heads`: Creates a list of multiple `Head` instances, each representing an individual attention head.

- **self.proj**: Projects the concatenated outputs from all attention heads back to the embedding dimension (`n_embd`).
 - **self.dropout**: Applies dropout after concatenating the outputs from all the heads and before passing them through the projection layer.
-

6. Forward Pass for **MultiHeadAttention** Class

```
def forward(self, x):
```

```
    out = torch.cat([h(x) for h in self.heads], dim=-1)
```

```
    out = self.dropout(self.proj(out))
```

```
    return out
```

- **out = torch.cat([h(x) for h in self.heads], dim=-1)**: Concatenates the outputs of all attention heads along the last dimension (feature dimension).
 - **out = self.dropout(self.proj(out))**: Passes the concatenated output through a linear projection and applies dropout.
-

7. **FeedForward** Class: Simple Feedforward Network

```
class FeedFoward(nn.Module):
```

```
    """ a simple linear layer followed by a non-linearity """
```

- The **FeedFoward** class defines a simple feedforward network that consists of two linear layers with a ReLU activation in between. This is used to introduce non-linearity in the model.
-

8. Initialization (**__init__**) Method for **FeedForward**

```
def __init__(self, n_embd):
```

```
    super().__init__()
```

```
self.net = nn.Sequential(
    nn.Linear(n_embd, 4 * n_embd),
    nn.ReLU(),
    nn.Linear(4 * n_embd, n_embd),
    nn.Dropout(dropout),
)
```

- **self.net**: Defines a sequence of layers: a linear layer that increases the embedding dimension by a factor of 4, a ReLU activation, another linear layer that reduces it back to **n_embd**, and a dropout layer for regularization.
-

9. Forward Pass for **FeedForward** Class

```
def forward(self, x):
    return self.net(x)
```

- The forward pass simply applies the defined sequence of layers to the input tensor **x**.
-

10. **Block** Class: Transformer Block

```
class Block(nn.Module):
    """ Transformer block: communication followed by computation """
```

- The **Block** class defines a single transformer block, which consists of two main parts:
 1. **Self-Attention**: The **MultiHeadAttention** mechanism.
 2. **FeedForward**: The feedforward network that applies a non-linearity.
-

11. Initialization (**__init__**) Method for **Block**

```

def __init__(self, n_embd, n_head):
    super().__init__()

    head_size = n_embd // n_head

    self.sa = MultiHeadAttention(n_head, head_size)

    self.ffwd = FeedFoward(n_embd)

    self.ln1 = nn.LayerNorm(n_embd)

    self.ln2 = nn.LayerNorm(n_embd)

```

- **self.sa**: Initializes the multi-head attention layer.
 - **self.ffwd**: Initializes the feedforward network.
 - **self.ln1**, **self.ln2**: Layer normalization layers that help stabilize training by normalizing activations.
-

12. Forward Pass for **Block** Class

```

def forward(self, x):

    y = self.sa(x)

    x = self.ln1(x + y)

    y = self.ffwd(x)

    x = self.ln2(x + y)

    return x

```

- **x + y**: Implements residual connections, adding the input **x** to the output of both the attention and feedforward layers.
 - **self.ln1(x + y)**: Applies layer normalization after the attention layer.
 - **self.ln2(x + y)**: Applies layer normalization after the feedforward network.
-

13. **GPTLanguageModel** Class: GPT Model


```

class GPTLanguageModel(nn.Module):

    def __init__(self, vocab_size):

        super().__init__()

        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)

        self.position_embedding_table = nn.Embedding(block_size, n_embd)

        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])

        self.ln_f = nn.LayerNorm(n_embd)

        self.lm_head = nn.Linear(n_embd, vocab_size)

        self.apply(self._init_weights)

```

- **self.token_embedding_table**: Embedding layer for token IDs.
- **self.position_embedding_table**: Embedding layer for position indices.
- **self.blocks**: A sequence of transformer blocks.
- **self.ln_f**: Final layer normalization.
- **self.lm_head**: A linear layer for output logits.

14. Forward Pass for **GPTLanguageModel**

```

def forward(self, index, targets=None):

    tok_emb = self.token_embedding_table(index) # (B,T,C)

    pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)

    x = tok_emb + pos_emb # (B,T,C)

    x = self.blocks(x) # (B,T,C)

    x = self.ln_f(x) # (B,T,C)

    logits = self.lm_head(x) # (B,T,vocab_size)

```

- **tok_emb**: Token embeddings for the input indices.
 - **pos_emb**: Positional embeddings for the input sequence length.
 - **x = tok_emb + pos_emb**: Adds token and position embeddings together.
 - **x = self.blocks(x)**: Passes the embeddings through the transformer blocks.
 - **logits = self.lm_head(x)**: Computes logits for each token position.
-

PART-7

1. Creating the Optimizer

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

- **optimizer**: Here, we create an optimizer using the **AdamW** algorithm. This is a variant of the Adam optimizer with weight decay (L2 regularization). AdamW helps improve the model's generalization by regularizing the weights during training.
 - **model.parameters()**: Specifies the parameters of the model that the optimizer will update during training.
 - **lr=learning_rate**: The learning rate determines the step size for the optimizer to update the model's parameters. It is set to the **learning_rate** variable.
-

2. Training Loop

```
for iter in range(max_iters):
```

```
    print(iter)
```

- **for iter in range(max_iters)**: This loop runs for a specified number of iterations (**max_iters**). Each iteration corresponds to a training step, where the model's parameters are updated.
 - **print(iter)**: Prints the current iteration number.
-

3. Model Evaluation and Logging Loss

```
if iter % eval_iters == 0:
```

```
losses = estimate_loss()
```

```
print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")
```

- **if iter % eval_iters == 0**: Every **eval_iters** iterations (i.e., periodic evaluation), the model's performance is evaluated.
 - **losses = estimate_loss()**: Calls the **estimate_loss()** function, which evaluates the model's performance on both the training and validation datasets. It returns the mean losses for both datasets.
 - **print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")**: Prints the current iteration and the corresponding training and validation losses.
-

4. Sampling a Batch of Data

```
xb, yb = get_batch('train')
```

- **xb, yb = get_batch('train')**: Samples a batch of data from the training set. **xb** represents the input sequence, and **yb** represents the corresponding target sequence.
-

5. Model Forward Pass and Loss Calculation

```
logits, loss = model.forward(xb, yb)
```

- **logits, loss = model.forward(xb, yb)**: Passes the input batch **xb** and target batch **yb** through the model's forward pass. The model returns:
 - **logits**: The raw predictions (logits) for each token in the sequence.
 - **loss**: The computed loss (cross-entropy loss) between the predicted logits and the true target sequence **yb**.
-

6. Optimizing the Model

```
optimizer.zero_grad(set_to_none=True)
```

```
loss.backward()
```

```
optimizer.step()
```

- **optimizer.zero_grad(set_to_none=True)**: Clears the gradients of the model's parameters to avoid accumulation of gradients from previous iterations. This is important as PyTorch accumulates gradients by default.
 - **loss.backward()**: Computes the gradients of the loss with respect to the model's parameters. This is done through backpropagation.
 - **optimizer.step()**: Updates the model's parameters based on the computed gradients and the learning rate.
-

7. Printing the Final Loss

```
print(loss.item())
```

- **print(loss.item())**: After completing the training loop, this line prints the final loss value (as a scalar) from the last iteration. The **item()** method converts the loss from a tensor to a Python number.

Summary

- **Optimizer Creation**: The AdamW optimizer is used for training, with model parameters being updated using backpropagation.
- **Training Loop**: For each iteration, the model is trained using a batch of data, the loss is calculated, and the optimizer updates the model's weights.
- **Periodic Evaluation**: Every few iterations (as specified by **eval_iters**), the model's performance is evaluated on the training and validation sets.
- **Gradient Updates**: Gradients are cleared, computed, and applied to the model parameters to minimize the loss function.

PART-8

1. Preparing the Input Prompt

```
prompt = 'Hello! Can you see me?'
```

```
context = torch.tensor(encode(prompt), dtype=torch.long, device=device)
```

- **prompt**: This is the initial string (the seed text) provided to the model. The goal is to generate a continuation of this prompt.
 - **encode(prompt)**: The **encode** function converts the input string (**prompt**) into a list of integers, where each integer corresponds to a character in the prompt. This is done by looking up the character's index from the **string_to_int** mapping.
 - **torch.tensor(..., dtype=torch.long, device=device)**: The list of integers is then converted into a PyTorch tensor and moved to the appropriate device (CPU or GPU) based on the **device** variable.
-

2. Generating New Tokens

```
generated_chars = decode(m.generate(context.unsqueeze(0),  
max_new_tokens=100)[0].tolist())
```

- **context.unsqueeze(0)**: The **context** tensor is reshaped to have an extra batch dimension. This is required because the model expects a batch of sequences, even if the batch size is 1.
 - **m.generate(context.unsqueeze(0), max_new_tokens=100)**: This line generates new tokens from the model (**m**) using the provided **context** (the prompt). The **generate** method runs the model in an autoregressive manner, generating one token at a time. The **max_new_tokens=100** argument specifies that a maximum of 100 new tokens should be generated.
 - The model generates one token at a time, appending it to the **context**, and updates the context with each new token.
 - **[0]**: Since the **generate** method returns a batch of sequences, the **[0]** extracts the first (and in this case, only) sequence from the batch.
 - **.tolist()**: Converts the tensor of generated token indices back to a list of integers.
 - **decode(...)**: The **decode** function takes the list of token indices and converts them back into the corresponding string representation.
-

3. Printing the Generated Text

```
print(generated_chars)
```

- **print(generated_chars):** The generated text (the original prompt followed by the model's output) is printed to the console.

Summary

- **Input Preparation:** The input prompt is encoded into token indices and passed to the model as context.
- **Text Generation:** The model generates a continuation of the input prompt, producing up to 100 new tokens.
- **Decoding:** The generated token indices are decoded back into a human-readable string.
- **Output:** The final generated text is printed, showing the model's attempt at continuing the input prompt.

Transformer Model Using the Mechanism: Self-Attention

Overview of Self-Attention in Transformers

Self-attention is a crucial mechanism in the transformer model, used to capture dependencies between words in a sentence, regardless of their distance from each other. The self-attention mechanism is used in multi-head attention, which is the foundation of transformer-based models like GPT (Generative Pre-trained Transformer).

Initial Model Behavior

In the early stages of training, the model has no context and is essentially "clueless." It explores random directions, attempting to converge and learn the best way to produce meaningful outputs. The model tunes parameters in the feed-forward network, multi-head attention, and normalization steps to eventually generate contextually accurate and coherent text.

Pretraining Process

During pretraining, the model is given a large set of inputs and learns to generate outputs by predicting probabilities for each token based on the previous tokens. Attention mechanisms help the model assign different attention scores to each token in the input sequence, which are influenced by the token's position and contextual relevance.

Types of Tokens

- **Character-level tokens:** Each character in the text is treated as a separate token.
- **Subword-level tokens:** Parts of words (e.g., prefixes or suffixes) are treated as tokens.
- **Word-level tokens:** Entire words are treated as tokens.

These tokens are mapped to vectors (embeddings) that represent the meaning of each token. The embedding vector for a common character like "E" might have a high frequency of usage, while less common characters like "Z" will have lower frequency embeddings. These embeddings are learned during training.

Input and Output

The input tokens are processed, and the output predictions are shifted by one token position to predict the next token in the sequence.

Layers in Transformer Model

A transformer model consists of several layers of encoders and decoders:

1. **Encoder layers:** These layers process the input tokens and generate an intermediate representation.
2. **Decoder layers:** These layers take the output of the encoder layers and generate the final predictions.

During training, the model processes the input through the encoder layers, and once the last encoder layer is reached, the outputs are passed to the decoder layers. The decoder layers apply transformations and generate probabilities, which are then used to sample the next token in the sequence.

Feed-Forward Network

Each transformer block contains a simple feed-forward network, which consists of:

- A linear layer
- A ReLU activation function
- Another linear layer

The purpose of this network is to process the input through transformations and add non-linearity, allowing the model to learn complex relationships in the data.

Residual Connections

A key feature in transformers is the use of residual connections:

- In these connections, the input is passed through a function (e.g., the feed-forward network or attention mechanism).
- The original input is added back to the output of the function, and the result is normalized.
- This helps avoid vanishing gradients and allows the model to retain useful information throughout deep networks.

Residual connections are critical in preventing the model from "forgetting" information as it moves through multiple layers, especially in deep networks.

Normalization Architectures

There are two common normalization strategies:

- **Pre-norm:** Normalization is applied before adding the residual connection.
- **Post-norm:** Residual connection is applied first, then normalization.

Post-norm is typically more effective in practice.

Multi-Head Attention

The key concept behind multi-head attention is parallelism. Instead of using a single attention mechanism, the transformer model uses multiple attention mechanisms (heads) in parallel, each of which can focus on different parts of the input sequence.

Analogy: Multiple Perspectives

Imagine ten different people reading the same book (e.g., *Harry Potter*). Each person interprets the content differently based on their cognitive abilities, experiences, and perspectives. Similarly, in multi-head attention, each "head" processes the input sequence from a different perspective, allowing the model to capture a wide range of information from the input.

- **Concatenation:** The outputs of the attention heads are concatenated.
- **Linear Transformation:** A final linear transformation is applied to combine the information from all heads into a single vector.

Self-Attention Mechanism

Self-attention helps the model decide which tokens in a sequence should be focused on. The mechanism uses three components:

- **Keys:** Represent the content of each token.
- **Queries:** Represent the question or focus of attention for each token.
- **Values:** Represent the information that needs to be aggregated.

The attention score is computed by taking the dot product of the query and key vectors. If the word "server" appears in two different contexts, the model will pay more attention to the relevant context by assigning higher attention scores to certain tokens.

Scaled Dot-Product Attention

The attention scores are scaled by the inverse square root of the key's length to prevent excessively large values, which can cause instability in training.

Masked Attention

To prevent the model from "cheating" by looking at future tokens, masked attention ensures that the model only attends to tokens in the past (or present). This is crucial for autoregressive models, where each token's prediction is based only on the tokens that came before it.

Softmax Function

The softmax function is applied to the attention scores to convert them into a probability distribution. This increases the model's confidence in attending to certain tokens over others.

Use of Encoders

Encoders are responsible for learning the dependencies between tokens in the input sequence. Through linear transformations, the encoder compresses the learned information into a manageable vector representation that is passed to the decoder.

Generative Pre-trained Transformer (GPT)

In GPT, the transformer architecture is simplified by using only decoder blocks, eliminating the need for multi-head attention in the encoder. This results in a model that generates text from a given prompt by predicting the next token in a sequence.

ENTIRE STEPS OF TRAINING AN LLM :

1. Library Imports

```
import torch
```

```
import torch.nn as nn
```

```
from torch.nn import functional as F
```

```
import mmap
```

```
import random
```

```
import pickle
```

```
import argparse
```

- `import torch`: This imports the main PyTorch library, which provides functions for tensor operations, deep learning, and model training.
- `import torch.nn as nn`: Imports the submodule `nn` from PyTorch, which contains classes and functions for building neural networks, such as layers (e.g., `Linear`,

`Embedding`), activation functions, and loss functions.

- `from torch.nn import functional as F`: Imports functional operations from `nn`, such as activation functions (e.g., `relu()`, `softmax()`) and loss functions (e.g., `cross_entropy()`).
 - `import mmap`: Imports the `mmap` module, which allows memory-mapped file access. This is helpful for reading large files in chunks without loading the entire file into memory.
 - `import random`: Imports Python's built-in `random` module, used for generating random numbers (in this case, for selecting random positions in the dataset).
 - `import pickle`: Imports the `pickle` module, which allows you to serialize Python objects (like models) for storage or transfer.
 - `import argparse`: Imports the `argparse` module, which provides functionality for parsing command-line arguments.
-

2. Argument Parsing

```
parser = argparse.ArgumentParser(description='This is a demonstration program')
```

- `argparse.ArgumentParser()`: Creates an `ArgumentParser` object that will handle parsing command-line arguments. The `description` parameter provides a description of what the program does.

```
parser.add_argument('-batch_size', type=str, required=True, help='Please provide a batch_size')
```

- `parser.add_argument()`: This line adds a command-line argument for the batch size. It expects an argument named `-batch_size` of type string (`type=str`). The `required=True` flag ensures that the argument must be provided when running the script. The `help` parameter provides a description of the argument that will be shown if the user requests help.

```
args = parser.parse_args()
```

- `parser.parse_args()`: This line parses the command-line arguments and stores them in the `args` variable.

```
print(f'batch size: {args.batch_size}')
```

- `print(f'batch size: {args.batch_size}')`: This prints the batch size provided by the user. The batch size is accessed from the parsed arguments (`args.batch_size`).
-

3. Device Configuration

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

- This line checks if a GPU is available (`torch.cuda.is_available()`).
 - If a GPU is available, it sets `device` to `'cuda'` (CUDA refers to GPU support in PyTorch).
 - If a GPU is not available, it sets `device` to `'cpu'` (using the CPU for computations).
-

4. Hyperparameters

```
batch_size = int(args.batch_size)
```

```
block_size = 128
```

```
max_iters = 200
```

```
learning_rate = 3e-4
```

```
eval_iters = 100
```

```
n_embd = 384
```

```
n_head = 1
```

```
n_layer = 1
```

```
dropout = 0.2
```

- `batch_size = int(args.batch_size)`: Converts the batch size provided by the user (as a string) to an integer.
- `block_size = 128`: Defines the size of the input block, which is the length of the text sequence the model will process in each batch.
- `max_iters = 200`: Sets the maximum number of iterations (or steps) for training the model.
- `learning_rate = 3e-4`: Defines the learning rate for the optimizer. It controls how much the model weights are updated during training.
- `eval_iters = 100`: Specifies how many iterations to run when evaluating the model during training.
- `n_embd = 384`: Sets the dimensionality of the embedding vectors (size of the word vectors).
- `n_head = 1`: Specifies the number of attention heads in the multi-head attention mechanism.
- `n_layer = 1`: Specifies the number of transformer blocks (layers) in the model.
- `dropout = 0.2`: Sets the dropout rate, which helps prevent overfitting by randomly setting a fraction of the activations to zero during training.

5. Vocabulary Preparation

```
chars = ""
```

```
with open("openwebtext/vocab.txt", 'r', encoding='utf-8') as f:
```

```
    text = f.read()
```

```
    chars = sorted(list(set(text)))
```

- `with open("openwebtext/vocab.txt", 'r', encoding='utf-8') as f:`
Opens the vocabulary file (`vocab.txt`) in read mode (`'r'`), using UTF-8 encoding.
- `text = f.read():` Reads the entire contents of the vocabulary file into the `text` variable.
- `chars = sorted(list(set(text))):` Converts the text into a set of unique characters, then sorts them alphabetically. This results in a list of unique characters in the vocabulary.

```
vocab_size = len(chars)
```

- `vocab_size = len(chars):` Calculates the size of the vocabulary by counting the number of unique characters in `chars`.

```
string_to_int = { ch:i for i,ch in enumerate(chars) }
```

```
int_to_string = { i:ch for i,ch in enumerate(chars) }
```

- `string_to_int:` Creates a dictionary that maps each character to a unique integer.
- `int_to_string:` Creates a dictionary that maps each integer back to its corresponding character.

```
encode = lambda s: [string_to_int[c] for c in s]
```

```
decode = lambda l: "".join([int_to_string[i] for i in l])
```

- `encode:` A lambda function that encodes a string `s` into a list of integers using the `string_to_int` dictionary.
- `decode:` A lambda function that decodes a list of integers `l` back into a string using the `int_to_string` dictionary.

6. Data Loading (Memory-mapped File Access)

```
def get_random_chunk(split):
```

```
filename = "openwebtext/train_split.txt" if split == 'train' else "openwebtext/val_split.txt"
```

```
with open(filename, 'rb') as f:
```

```
    with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
```

```
        file_size = len(mm)
```

```
        start_pos = random.randint(0, (file_size) - block_size*batch_size)
```

```
        mm.seek(start_pos)
```

```
        block = mm.read(block_size*batch_size-1)
```

```
        decoded_block = block.decode('utf-8', errors='ignore').replace('\r', "")
```

```
        data = torch.tensor(encode(decoded_block), dtype=torch.long)
```

```
    return data
```

- `get_random_chunk(split)`: This function loads a random chunk of text from a file. The `split` argument determines whether to load the training or validation data.
- `filename = "openwebtext/train_split.txt" if split == 'train' else "openwebtext/val_split.txt"`: Selects the appropriate file based on the `split` argument.
- `with open(filename, 'rb') as f`: Opens the file in binary read mode ('rb').
- `with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm`: Maps the entire file into memory. This allows for efficient access to large files without loading them completely into memory.
- `file_size = len(mm)`: Calculates the file size to determine the range for random access.
- `start_pos = random.randint(0, (file_size) - block_size*batch_size)`: Selects a random position to start reading the chunk of text, ensuring the chunk fits within the file.
- `mm.seek(start_pos)`: Moves the file pointer to the random starting position.

- `block = mm.read(block_size*batch_size-1)`: Reads the chunk of text from the memory-mapped file.
 - `decoded_block = block.decode('utf-8', errors='ignore').replace('\r', ' ')`: Decodes the byte sequence into a string and removes any carriage return characters.
 - `data = torch.tensor(encode(decoded_block), dtype=torch.long)`: Encodes the decoded block of text into integers and converts it into a PyTorch tensor.
-

7. Get Batch Function

```
def get_batch(split):
```

```
    data = get_random_chunk(split)
```

```
    ix = torch.randint(len(data) - block_size, (batch_size,))
```

```
    x = torch.stack([data[i:i+block_size] for i in ix])
```

```
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
```

```
    x, y = x.to(device), y.to(device)
```

```
    return x, y
```

- `def get_batch(split)`: This function retrieves a batch of data for training or validation, based on the `split` argument (which can either be `'train'` or `'val'`).
- `data = get_random_chunk(split)`: Fetches a random chunk of text data from the corresponding file (train or validation) using the `get_random_chunk()` function defined earlier.
- `ix = torch.randint(len(data) - block_size, (batch_size,))`: Generates `batch_size` random indices from the `data` tensor, ensuring that each index is positioned such that the following `block_size` elements can be read from it.
- `x = torch.stack([data[i:i+block_size] for i in ix])`: Creates the input batch `x` by stacking slices of `data`, each of size `block_size`, starting from the indices

`ix.`

- `y = torch.stack([data[i+1:i+block_size+1] for i in ix])`: Creates the target batch `y` by shifting the slices by 1. This ensures that the model learns to predict the next character in the sequence.
- `x, y = x.to(device), y.to(device)`: Moves both `x` and `y` tensors to the device (`cuda` if GPU is available, else `cpu`).
- `return x, y`: Returns the input batch `x` and the target batch `y`.

8. Loss Estimation Function

`@torch.no_grad()`

`def estimate_loss():`

`out = {}`

`model.eval()`

`for split in ['train', 'val']:`

`losses = torch.zeros(eval_iters)`

`for k in range(eval_iters):`

`X, Y = get_batch(split)`

`logits, loss = model(X, Y)`

`losses[k] = loss.item()`

`out[split] = losses.mean()`

`model.train()`

`return out`

- `@torch.no_grad()`: A decorator that tells PyTorch not to compute gradients within this function, saving memory and computations during evaluation.
- `def estimate_loss()`: Defines the function that estimates the loss on both the training and validation splits.
- `out = {}`: Initializes an empty dictionary to store the average losses for each data split.
- `model.eval()`: Sets the model to evaluation mode, which turns off behaviors like dropout and batch normalization that are only needed during training.
- `for split in ['train', 'val']`: Iterates over the training and validation splits.
- `losses = torch.zeros(eval_iters)`: Initializes a tensor to store the loss values for each evaluation iteration.
- `for k in range(eval_iters)`: Loops over a fixed number of evaluation iterations.
- `X, Y = get_batch(split)`: Fetches a batch of data (inputs and targets) from the current split.
- `logits, loss = model(X, Y)`: Computes the model's predictions (`logits`) and the loss between the predictions and true targets.
- `losses[k] = loss.item()`: Stores the current loss in the `losses` tensor.
- `out[split] = losses.mean()`: Computes and stores the mean of the losses for the current split.
- `model.train()`: Sets the model back to training mode after the evaluation.
- `return out`: Returns a dictionary containing the average losses for the training and validation sets.

9. Self-Attention Mechanism (Head Class)

```
class Head(nn.Module):
```

```
    """ one head of self-attention """
```

- `class Head(nn.Module)`: Defines a class `Head` that represents a single head in the multi-head self-attention mechanism.

```
def __init__(self, head_size):
```

```
    super().__init__()
```

```
    self.key = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.query = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.value = nn.Linear(n_embd, head_size, bias=False)
```

```
    self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
```

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

- `def __init__(self, head_size)`: The constructor initializes the attention head with the specified `head_size`.
- `self.key = nn.Linear(n_embd, head_size, bias=False)`: Creates a linear layer for the key transformation, converting the input embeddings (`n_embd`) to the head size (`head_size`).
- `self.query = nn.Linear(n_embd, head_size, bias=False)`: Creates a linear layer for the query transformation, similarly mapping the input embeddings to the head size.
- `self.value = nn.Linear(n_embd, head_size, bias=False)`: Creates a linear layer for the value transformation.
- `self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))`: Registers a lower triangular matrix (`tril`) to apply a mask during attention computation, ensuring that each token only attends to previous tokens (for causal self-attention).
- `self.dropout = nn.Dropout(dropout)`: Initializes a dropout layer with the specified dropout rate to prevent overfitting.

```

def forward(self, x):

    B, T, C = x.shape

    k = self.key(x) # (B,T,hs)

    q = self.query(x) # (B,T,hs)

    wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B, hs, T) -> (B, T, T)

    wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)

    wei = F.softmax(wei, dim=-1) # (B, T, T)

    wei = self.dropout(wei)

    v = self.value(x) # (B,T,hs)

    out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)

    return out

```

- `def forward(self, x):` Defines the forward pass of the attention head. The input `x` has the shape `(batch_size, sequence_length, embedding_dim)`.
- `k = self.key(x), q = self.query(x), v = self.value(x):` Computes the key, query, and value vectors by passing the input `x` through their respective linear layers.
- `wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5:` Computes the attention scores by performing a matrix multiplication between the query `q` and the transposed key `k`. The result is scaled by the square root of the key size (`head_size`).
- `wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')):` Applies a mask to the attention scores, ensuring that each token can only attend to previous tokens (this prevents future information from leaking into the model).
- `wei = F.softmax(wei, dim=-1):` Applies the softmax function to the attention scores, converting them into probabilities.
- `wei = self.dropout(wei):` Applies dropout to the attention weights for regularization.

- `out = wei @ v`: Performs a weighted sum of the value vectors, using the attention weights `wei` as the coefficients.
- `return out`: Returns the output of the attention head.

10. Multi-Head Attention Mechanism

`class MultiHeadAttention(nn.Module):`

`""" multiple heads of self-attention in parallel """`

- `class MultiHeadAttention(nn.Module)`: Defines a class `MultiHeadAttention`, which combines multiple `Head` objects in parallel.

`def __init__(self, num_heads, head_size):`

`super().__init__()`

`self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])`

`self.proj = nn.Linear(head_size * num_heads, n_embd)`

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

- `self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])`: Creates a list of attention heads (each an instance of the `Head` class), where `num_heads` determines how many heads to use.
- `self.proj = nn.Linear(head_size * num_heads, n_embd)`: Defines a linear projection layer that projects the concatenated outputs of all heads back to the original embedding dimension `n_embd`.
- `self.dropout = nn.Dropout(dropout)`: Initializes a dropout layer for regularization.

`def forward(self, x):`

```
out = torch.cat([h(x) for h in self.heads], dim=-1) # (B, T, F) -> (B, T, [h1, h1, h1, h1, h2, h2, h2, h2, h3, h3, h3, h3])
```

```
out = self.dropout(self.proj(out))
```

```
return out
```

- `

`out = torch.cat([h(x) for h in self.heads], dim=-1)`: Passes the input `x`` through all attention heads and concatenates their outputs along the last dimension (the embedding dimension).

- `out = self.dropout(self.proj(out))`: Projects the concatenated outputs to the original embedding dimension and applies dropout for regularization.
- `return out`: Returns the output after the multi-head attention mechanism.

12. Feed-Forward Network

```
class FeedForward(nn.Module):
```

```
    """ a simple linear layer followed by a non-linearity """
```

- `class FeedForward(nn.Module)`: Defines a class `FeedForward`, which implements a simple feed-forward neural network used in transformer blocks after the attention mechanism.

```
def __init__(self, n_embd):
```

```
    super().__init__()
```

```
    self.net = nn.Sequential(
```

```
        nn.Linear(n_embd, 4 * n_embd),
```

```
        nn.ReLU(),
```

```
        nn.Linear(4 * n_embd, n_embd),
```

```
        nn.Dropout(dropout),
```

)

- `def __init__(self, n_embd)`: The constructor initializes the feed-forward network with the given embedding size `n_embd`.
- `self.net = nn.Sequential(...)`: Creates a sequential model with three layers:
 - `nn.Linear(n_embd, 4 * n_embd)`: A linear layer that maps the input to a 4 times larger dimension (`4 * n_embd`), increasing the capacity of the network.
 - `nn.ReLU()`: A ReLU activation function introduces non-linearity to the model.
 - `nn.Linear(4 * n_embd, n_embd)`: A linear layer that maps the output of the previous layer back to the original embedding dimension `n_embd`.
 - `nn.Dropout(dropout)`: A dropout layer is applied after the feed-forward network for regularization.

```
def forward(self, x):
```

```
    return self.net(x)
```

- `def forward(self, x)`: The forward pass through the feed-forward network. It simply applies the sequential layers to the input `x` and returns the output.

13. Transformer Block

```
class Block(nn.Module):
```

```
    """ Transformer block: communication followed by computation """
```

- `class Block(nn.Module)`: Defines a transformer block, which consists of a self-attention mechanism and a feed-forward network, with layer normalization applied after each.

```
def __init__(self, n_embd, n_head):
```

```
    super().__init__()
```

```
    head_size = n_embd // n_head
```

```
self.sa = MultiHeadAttention(n_head, head_size)
```

```
self.ffwd = FeedForward(n_embd)
```

```
self.ln1 = nn.LayerNorm(n_embd)
```

```
self.ln2 = nn.LayerNorm(n_embd)
```

- `def __init__(self, n_embd, n_head)`: The constructor initializes the transformer block with `n_embd` as the embedding size and `n_head` as the number of attention heads.
- `head_size = n_embd // n_head`: Computes the size of each individual attention head by dividing the embedding size by the number of heads.
- `self.sa = MultiHeadAttention(n_head, head_size)`: Initializes a multi-head attention mechanism using the computed head size and number of heads.
- `self.ffwd = FeedForward(n_embd)`: Initializes the feed-forward network with the embedding size.
- `self.ln1 = nn.LayerNorm(n_embd)`: Applies layer normalization after the self-attention mechanism.
- `self.ln2 = nn.LayerNorm(n_embd)`: Applies layer normalization after the feed-forward network.

```
def forward(self, x):
```

```
    y = self.sa(x)
```

```
    x = self.ln1(x + y)
```

```
    y = self.ffwd(x)
```

```
    x = self.ln2(x + y)
```

```
    return x
```

- `def forward(self, x)`: Defines the forward pass through the transformer block.
 - `y = self.sa(x)`: The input `x` is passed through the multi-head attention mechanism (`self.sa`).
 - `x = self.ln1(x + y)`: The residual connection (`x + y`) is applied, followed by layer normalization. This helps prevent vanishing gradients and stabilizes training.
 - `y = self.ffwd(x)`: The output from the previous step (`x`) is passed through the feed-forward network (`self.ffwd`).
 - `x = self.ln2(x + y)`: Another residual connection (`x + y`) is applied, followed by layer normalization.
 - `return x`: The output `x` is returned after passing through the attention and feed-forward layers.
-

14. GPT Language Model

```
class GPTLanguageModel(nn.Module):
```

```
    def __init__(self, vocab_size):
```

```
        super().__init__()
```

```
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
```

```
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
```

```
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
```

```
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
```

```
        self.lm_head = nn.Linear(n_embd, vocab_size)
```

- `class GPTLanguageModel(nn.Module)`: Defines the GPT (Generative Pretrained Transformer) language model, which is built using transformer blocks.

- `self.token_embedding_table = nn.Embedding(vocab_size, n_embd):` Initializes the token embedding layer. This layer converts the token IDs into dense vectors of size `n_embd`.
- `self.position_embedding_table = nn.Embedding(block_size, n_embd):` Initializes the position embedding layer, which encodes the position of each token in the input sequence (since transformers do not inherently understand token order).
- `self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)]):` Creates a stack of `n_layer` transformer blocks. Each block is initialized with the given embedding size `n_embd` and number of heads `n_head`.
- `self.ln_f = nn.LayerNorm(n_embd):` A final layer normalization to stabilize the output of the transformer blocks.
- `self.lm_head = nn.Linear(n_embd, vocab_size):` A linear layer that projects the output of the transformer blocks to the vocabulary size, making it suitable for the final language modeling task.

```
def _init_weights(self, module):
```

```
    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 _init_weights(self, module):` This function initializes the weights of the model. It uses the normal distribution with a mean of 0 and a standard deviation of 0.02 for both `nn.Linear` and `nn.Embedding` layers.

```
def forward(self, index, targets=None):
```

```
    print(index.shape)
```

```
    B, T = index.shape
```

```

tok_emb = self.token_embedding_table(index) # (B,T,C)

pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)

x = tok_emb + pos_emb # (B,T,C)

x = self.blocks(x) # (B,T,C)

x = self.ln_f(x) # (B,T,C)

logits = self.lm_head(x) # (B,T,vocab_size)

```

if targets is None:

```

    loss = None

```

else:

```

    B, T, C = logits.shape

    logits = logits.view(B*T, C)

    targets = targets.view(B*T)

    loss = F.cross_entropy(logits, targets)

```

return logits, loss

- `def forward(self, index, targets=None)`: The forward pass of the GPT model.
 - `tok_emb = self.token_embedding_table(index)`: The input tokens are converted into token embeddings using the token embedding table.
 - `pos_emb = self.position_embedding_table(torch.arange(T, device=device))`: The position indices are converted into position embeddings.

- `x = tok_emb + pos_emb`: The token embeddings and position embeddings are added together, forming the final input to the transformer blocks.
- `x = self.blocks(x)`: The input `x` is passed through the stack of transformer blocks.
- `x = self.ln_f(x)`: Layer normalization is applied to the output of the transformer blocks.
- `logits = self.lm_head(x)`: The final output is projected to the vocabulary size to make predictions for each token in the sequence.
- If `targets` is provided, the loss is computed using cross-entropy between the predicted logits and the true target tokens.
- `return logits, loss`: The function returns the logits (predictions) and the loss (if targets are provided).

15. Text Generation

def generate(self, index, max_new_tokens):

 # index is (B, T) array of indices in the current context

 for _ in range(max_new_tokens):

 logits, loss = self.forward(index)

 logits = logits[:, -1, :] # becomes (B, C)

 probs = F.softmax(logits, dim=-1) # (B, C)

 index_next = torch.multinomial(probs, num_samples=1) # (B, 1)

 index = torch.cat((index, index_next), dim=1) # (B, T+1)

 return index

- `def generate(self, index, max_new_tokens):` This function generates new tokens based on the current input sequence.
 - For each new token

to be generated (`max_new_tokens` times), the model generates the next token using the current sequence `index`.

- `logits, loss = self.forward(index):` Calls the forward pass to get the model's predictions (logits) for the next token.
- `logits = logits[:, -1, :]:` Focuses on the last time step of the sequence, which corresponds to the next token to be predicted.
- `probs = F.softmax(logits, dim=-1):` Applies softmax to the logits to convert them into probabilities.
- `index_next = torch.multinomial(probs, num_samples=1):` Samples a token from the probability distribution.
- `index = torch.cat((index, index_next), dim=1):` Appends the newly generated token to the current sequence.
- `return index:` After generating the required number of new tokens, the updated sequence is returned.

16. Training Loop

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

- `optimizer = torch.optim.AdamW(...):` Initializes the AdamW optimizer for training the model.

```
for iter in range(max_iters):
```

```
    print(iter)
```

```
    if iter % eval_iters == 0:
```

```
losses = estimate_loss()
```

```
print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")
```

- The loop runs for `max_iters` iterations, printing the current iteration number. Every `eval_iters` iterations, it estimates the loss on the training and validation sets and prints it.

```
xb, yb = get_batch('train')
```

- Samples a batch of data from the training set using `get_batch()`.

```
logits, loss = model.forward(xb, yb)
```

```
optimizer.zero_grad(set_to_none=True)
```

```
loss.backward()
```

```
optimizer.step()
```

- Passes the batch through the model to calculate the loss.
- Zeroes the gradients using `optimizer.zero_grad()`.
- Computes the gradients of the loss with respect to the model parameters using `loss.backward()`.
- Updates the model parameters with `optimizer.step()`.

```
print(loss.item())
```

- Prints the final loss for the current iteration.

```
with open('model-01.pkl', 'wb') as f:
```

```
pickle.dump(model, f)
```

- After training, the model is saved to a file named `model-01.pkl` using the `pickle` module.

```
print('model saved')
```

- Prints a message confirming the model has been saved.
-

MATH USED FOR MODEL TRAINING

1. Imports and Setup

```
import torch
import torch.nn as nn
from torch.nn import functional as F
import numpy as np
import time
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

- **torch**: The core PyTorch library, used for tensor operations and deep learning functionality.
- **torch.nn**: Contains neural network layers and utilities, such as Linear layers, loss functions, etc.
- **torch.nn.functional**: Provides functions (like softmax) without parameters, often used in models.
- **numpy**: A library for numerical computations, used alongside PyTorch in certain operations.
- **time**: Used for performance measurement (i.e., timing code execution).
- **device**: This checks if CUDA (GPU) is available for computation; if not, it defaults to CPU.
- **print(device)**: Prints the device being used, either 'cuda' or 'cpu'.

2. Measure Time for Matrix Operations

```
%%time
start_time = time.time()
zeros = torch.zeros(1, 1)
end_time = time.time()

elapsed_time = end_time - start_time
print(f"{elapsed_time:.8f}")
```

- **%%time**: Jupyter magic command to time the code execution.
- **torch.zeros(1, 1)**: Creates a tensor filled with zeros (1x1 matrix).
- **start_time** and **end_time**: Used to calculate the elapsed time of the operation.
- **elapsed_time**: Measures how long the operation takes.
- **print(f"{elapsed_time:.8f}")**: Prints the elapsed time with 8 decimal places.

3. Matrix Multiplication on CPU vs GPU

```
torch_rand1 = torch.rand(100, 100, 100, 100).to(device)
torch_rand2 = torch.rand(100, 100, 100, 100).to(device)
np_rand1 = torch.rand(100, 100, 100, 100)
np_rand2 = torch.rand(100, 100, 100, 100)
```

- **torch.rand()**: Creates a tensor with random values. The **to(device)** sends the tensor to the specified device (GPU or CPU).
- Two tensors (**torch_rand1** and **torch_rand2**) are created on the device, and two others (**np_rand1**, **np_rand2**) on CPU.

```
start_time = time.time()
rand = (torch_rand1 @ torch_rand2)
end_time = time.time()
elapsed_time = end_time - start_time
print(f"{elapsed_time:.8f}") # For GPU
```

- **torch_rand1 @ torch_rand2**: Performs matrix multiplication between the two tensors on the GPU.
- Measures and prints the time taken.

```
start_time = time.time()
rand = np.multiply(np_rand1, np_rand2)
end_time = time.time()
elapsed_time = end_time - start_time
print(f"{elapsed_time:.8f}") # For CPU
```

- **np.multiply()**: Multiplies tensors element-wise on the CPU using NumPy.
- Measures and prints the time taken.

4. Multinomial Sampling (Prediction Task)

```
probabilities = torch.tensor([0.1, 0.9])
samples = torch.multinomial(probabilities, num_samples=10, replacement=True)
print(samples)
```

- **torch.multinomial()**: Samples from a multinomial distribution with the given probabilities.
- This is useful for tasks like predicting the next word or character based on probabilities (like language modeling).
- The **replacement=True** allows for sampling the same element multiple times.

5. Concatenating Tensors

```
tensor = torch.tensor([1, 2, 3, 4])
out = torch.cat((tensor, torch.tensor([5])), dim=0)
out
```

- **torch.cat()**: Concatenates tensors along a specified dimension (**dim=0** means concatenation along the first axis).
- The result is a tensor that includes the original tensor and the added value **5**.

6. Lower and Upper Triangular Matrices

```
out = torch.tril(torch.ones(5, 5))
out
```

- **torch.tril()**: Creates a lower triangular matrix where all elements above the diagonal are set to 0.
- The result is a matrix filled with 1s below and on the diagonal, and 0s above the diagonal.

```
out = torch.triu(torch.ones(5, 5))
out
```

- **torch.triu()**: Creates an upper triangular matrix where all elements below the diagonal are set to 0.

7. Masked Fill

```
out = torch.zeros(5, 5).masked_fill(torch.tril(torch.ones(5, 5)) == 0, float('-inf'))
out
```

- **masked_fill()**: Fills elements of a tensor with a specified value where a mask condition is met.
- Here, it replaces all values in the lower triangle of the matrix with **-inf**.


```
torch.exp(out)
```

- **torch.exp()**: Applies the exponential function element-wise to the tensor. This is often used in softmax functions to make values more distinguishable.

8. Stacking Tensors

```
tensor1 = torch.tensor([1, 2, 3])  
tensor2 = torch.tensor([4, 5, 6])  
tensor3 = torch.tensor([7, 8, 9])  
stacked_tensor = torch.stack([tensor1, tensor2, tensor3])  
stacked_tensor
```

- **torch.stack()**: Stacks tensors along a new dimension. The result is a 2D tensor where each original tensor becomes a row.

9. Transposing Tensors

```
input = torch.zeros(2, 3, 4)  
out1 = input.transpose(0, 1)  
out2 = input.transpose(-2, -1)  
print(out1.shape)  
print(out2.shape)
```

- **torch.transpose()**: Swaps the dimensions of a tensor. `transpose(0, 1)` swaps the first and second dimensions, and `transpose(-2, -1)` swaps the last two dimensions.
- `print(out1.shape)` and `print(out2.shape)` display the resulting tensor shapes.

10. Permuting Tensors

```
torch.Size([3, 2, 4])  
torch.Size([2, 4, 3])
```

- **torch.permute()**: Changes the order of dimensions. In this case, the dimensions are reordered to different orders as specified.

11. Linear Layer

```
linear = nn.Linear(3, 3, bias=False)  
print(linear(sample))
```

- **nn.Linear()**: Defines a linear layer with input size 3 and output size 3. **bias=False** means there is no bias term in the layer.
- **linear(sample)**: Applies the linear transformation to the input tensor **sample**.

12. Softmax

```
softmax_output = F.softmax(tensor1, dim=0)
print(softmax_output)
```

- **F.softmax()**: Applies the softmax function to the tensor along the specified dimension (**dim=0**).
- The softmax function transforms the input tensor into a probability distribution.

13. Embedding Vectors

```
embedding = nn.Embedding(vocab_size, embedding_dim)
input_indices = torch.LongTensor([1, 5, 3, 2])
embedded_output = embedding(input_indices)
print(embedded_output.shape)
```

- **nn.Embedding()**: Defines an embedding layer for discrete indices. It maps each index to a dense vector representation.
- The output **embedded_output** represents the input indices as vectors in a higher-dimensional space.

14. Matrix Multiplication (Matmul)

```
print(torch.matmul(a, b))
```

- **torch.matmul()**: Performs matrix multiplication (a standard operation in neural networks for applying weights).

15. View and Reshape

```
a = torch.rand(2, 3, 5)
x, y, z = a.shape
a = a.view(x,y,z)
print(a.shape)
```

- **view()**: Reshapes the tensor without changing its data. This is useful for flattening or reordering dimensions.

16. View with Batch Processing

```
output = input.view(B*T, C)
print(output)
```

- This flattens a 3D tensor into a 2D tensor, useful for processing batches of data in neural networks.

The code covers various operations in PyTorch, including matrix operations, tensor reshaping, and neural network components. It also compares CPU vs GPU performance for matrix multiplication and demonstrates important neural network concepts like embedding layers, softmax, and linear transformations.

DATA EXTRACTION

Step-by-step Explanation of the Code:

Importing Required Libraries:

```
import os
import lzma # for handling xz files (compressed files)
import tqdm # for displaying a progress bar (moving from left to right)
```

- **os**: Provides functionalities for interacting with the operating system, such as file and directory operations.
- **lzma**: Used to handle xz-compressed files. It allows you to open, read, and decompress **.xz** files.
- **tqdm**: Provides a progress bar in the terminal, commonly used in loops for tracking progress, although it's not used after the **ModuleNotFoundError** was encountered.

Function Definition to Get **.xz** Files in a Directory:

```
def xz_files_in_dir(directory):
    files = []
    for filename in os.listdir(directory):
        if filename.endswith(".xz") and os.path.isfile(os.path.join(directory, filename)):
            files.append(filename)
    return files
```

- **xz_files_in_dir**: This function takes a directory path as an argument and returns a list of **.xz** files in that directory.
 - **os.listdir(directory)**: Lists all items (files and directories) in the specified directory.

- **os.path.isfile()**: Checks if the item is a file (not a directory).
- **filename.endswith(".xz")**: Filters the files to include only those with the .xz extension.

Variable Initialization:

```
folder_path = "D:\\openwebtext\\openwebtext"
output_file_train = "output_train.txt" # 90% of data
output_file_val = "output_val.txt" # 10% of data
vocab_file = "vocab.txt" # all new characters from the giant corpus will be pushed here.
```

- **folder_path**: The path where .xz files are located.
- **output_file_train**: File where the training data will be saved (90% of data).
- **output_file_val**: File where the validation data will be saved (10% of data).
- **vocab_file**: File where the unique characters found in the corpus will be written.

Retrieving and Splitting Files for Training and Validation:

```
files = xz_files_in_dir(folder_path) # list of file names stored in this variable
total_files = len(files)
```

```
# Calculate the split indices
split_index = int(total_files * 0.9) # 90% for training
files_train = files[:split_index]
files_val = files[split_index:]
```

- **files**: Stores the list of .xz files obtained from the directory.
- **total_files**: Stores the total number of .xz files.
- **split_index**: Splits the files into training and validation sets with 90% for training and 10% for validation.
- **files_train and files_val**: Store the file names for the training and validation datasets, respectively.

Vocabulary Initialization:

```
vocab = set() # set is a collection of unique items (here new chars in giant corpus)
```

- **vocab**: A set to store unique characters found in the corpus. A set is used because it only stores unique values.

Processing Training Files:

```
with open(output_file_train, "w", encoding="utf-8") as outfile:
    for filename in files_train:
```

```

file_path = os.path.join(folder_path, filename)
with lzma.open(file_path, "rt", encoding="utf-8") as infile:
    text = infile.read()
    outfile.write(text)
    characters = set(text)
    vocab.update(characters)

```

- **open(output_file_train, "w", encoding="utf-8")**: Opens the output file in write mode with UTF-8 encoding to store the training data.
- **for filename in files_train**: Iterates through the training files.
 - **os.path.join(folder_path, filename)**: Joins the folder path and filename to get the full path of the file.
 - **lzma.open(file_path, "rt", encoding="utf-8")**: Opens each **.xz** file in text mode ("rt") with UTF-8 encoding for reading.
 - **infile.read()**: Reads the entire content of the **.xz** file.
 - **outfile.write(text)**: Writes the content of the file to the **output_file_train**.
 - **characters = set(text)**: Converts the text into a set of unique characters.
 - **vocab.update(characters)**: Updates the **vocab** set with the unique characters from the current file.

Processing Validation Files:

```

with open(output_file_val, "w", encoding="utf-8") as outfile:
    for filename in files_val:
        file_path = os.path.join(folder_path, filename)
        with lzma.open(file_path, "rt", encoding="utf-8") as infile:
            text = infile.read()
            outfile.write(text)
            characters = set(text)
            vocab.update(characters)

```

- Similar to the training files, this block processes the validation files in the same way: reading, writing, and updating the vocabulary.

Writing Vocabulary to **vocab.txt**:

```

with open(vocab_file, "w", encoding="utf-8") as vfile:
    for char in vocab:
        vfile.write(char + '\n')

```

```

print("Vocabulary written to", vocab_file)

```

- `open(vocab_file, "w", encoding="utf-8")`: Opens the vocabulary file in write mode with UTF-8 encoding.
- `for char in vocab`: Iterates through each unique character in the `vocab` set.
- `vfile.write(char + '\n')`: Writes each character followed by a newline to the `vocab.txt` file.
- `print("Vocabulary written to", vocab_file)`: Prints a message confirming that the vocabulary has been written.

Issues Encountered:

The code originally included a progress bar using the `tqdm` library, but due to a `ModuleNotFoundError`, it was removed. As a result, the loop processing the training and validation files no longer has the visual progress bar.

BIGRAM MODEL

1. Importing Required Libraries

```
import torch.nn as nn
from torch.nn import functional as F
```

- `import torch.nn as nn`: This imports the `torch.nn` module, which contains essential tools for building neural networks in PyTorch. The alias `nn` is commonly used in the PyTorch community to access the neural network modules.
- `from torch.nn import functional as F`: The `functional` module provides a collection of functions like activation functions (e.g., ReLU, Sigmoid) and loss functions (e.g., cross-entropy loss) that don't require you to define a separate class. `F` is the commonly used alias for this module.

2. Device Setup

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

- `torch.cuda.is_available()`: This checks if a GPU is available for computation. CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model created by Nvidia, allowing the use of GPUs.
- `device = 'cuda' if torch.cuda.is_available() else 'cpu'`: This sets the `device` variable to `'cuda'` if a GPU is available, otherwise, it sets it to `'cpu'` for CPU computation.

- **print(device)**: This prints the chosen device ('cuda' or 'cpu') to confirm which device will be used for model training.

3. Hyperparameters

```
max_iters = 10000 # how many iterations we are going to have in the training loop
learning_rate = 3e-4
eval_iters = 250
block_size = 8 # length of integers (sequence length)
batch_size = 4 # number of sequences processed in parallel
```

- **max_iters = 10000**: This sets the number of training iterations (steps) the model will run. The model will train for 10,000 iterations.
- **learning_rate = 3e-4**: This is the learning rate for the optimizer, which controls how much to change the model's parameters with respect to the gradient. A smaller learning rate means the model updates slowly but more precisely.
- **eval_iters = 250**: This sets the number of iterations between each evaluation of the model on validation data. Every 250 iterations, the model will be evaluated.
- **block_size = 8**: The sequence length of the input data. In the context of a language model, this means the number of characters the model looks at when making predictions.
- **batch_size = 4**: This defines the number of sequences that will be processed in parallel during each iteration of training. Larger batches can speed up training but require more memory.

4. Reading and Preparing the Dataset

```
with open('wizard_of_oz.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

- **with open('wizard_of_oz.txt', 'r', encoding='utf-8') as f**: This opens the "wizard_of_oz.txt" file in read mode (' r ') with UTF-8 encoding. The **with** statement ensures that the file is automatically closed after reading.
- **text = f.read()**: Reads the entire content of the file into the **text** variable.

```
chars = sorted(set(text))
print(chars)
```

- **set(text)**: Converts the text into a set, which removes duplicate characters and returns only the unique characters present in the text.
- **sorted(set(text))**: Sorts the set of unique characters alphabetically.

- **print(chars)**: Prints the sorted list of unique characters found in the text.

```
vocab_size = len(chars)
```

- **vocab_size = len(chars)**: This calculates the number of unique characters in the **text** (i.e., the vocabulary size).

```
string_to_int = {ch: i for i, ch in enumerate(chars)}
```

```
int_to_string = {i: ch for i, ch in enumerate(chars)}
```

- **string_to_int = {ch: i for i, ch in enumerate(chars)}**: Creates a dictionary that maps each unique character (**ch**) to a unique integer (**i**).
- **int_to_string = {i: ch for i, ch in enumerate(chars)}**: This creates the reverse mapping, where each integer is mapped to its corresponding character.

```
encode = lambda s: [string_to_int[c] for c in s]
```

```
decode = lambda l: ''.join([int_to_string[i] for i in l])
```

- **encode = lambda s: [string_to_int[c] for c in s]**: Defines a function **encode** that takes a string **s** and returns a list of integers, where each character in the string is replaced by its corresponding integer from the **string_to_int** mapping.
- **decode = lambda l: ''.join([int_to_string[i] for i in l])**: Defines a function **decode** that takes a list of integers **l** and returns the corresponding string by mapping each integer to its corresponding character using **int_to_string**.

```
data = torch.tensor(encode(text), dtype=torch.long)
```

- **encode(text)**: Encodes the entire **text** into a list of integers.
- **torch.tensor(encode(text), dtype=torch.long)**: Converts the encoded list into a PyTorch tensor of type **torch.long**, which is suitable for training.

5. Training and Validation Split

```
n = int(0.8 * len(data))
```

```
train_data = data[:n]
```

```
val_data = data[n:]
```

- **n = int(0.8 * len(data))**: Splits the dataset into 80% for training and 20% for validation. Here, **n** is the number of training data points (80% of the total length).

- **train_data = data[:n]**: The first **n** data points are used for training.
- **val_data = data[n:]**: The remaining data points are used for validation.

6. Creating Data Batches

```
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i + block_size] for i in ix]) # stacks them in batches
    y = torch.stack([data[i + 1:i + block_size + 1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

- **def get_batch(split)::** This function generates a batch of data for training or validation based on the **split** argument.
- **data = train_data if split == 'train' else val_data:** Depending on whether **split** is 'train' or 'val', the function selects either the training or validation data.
- **ix = torch.randint(len(data) - block_size, (batch_size,)):** Randomly generates **batch_size** indices that will serve as starting points for the sequences.
- **x = torch.stack([data[i:i + block_size] for i in ix]):** For each index in **ix**, it selects a block of **block_size** consecutive data points to form the input sequences (**x**).
- **y = torch.stack([data[i + 1:i + block_size + 1] for i in ix]):** This creates the target sequences (**y**) by shifting each input sequence by one element (the next character in the sequence).
- **x, y = x.to(device), y.to(device):** Moves both **x** and **y** to the device (either GPU or CPU).
- **return x, y:** Returns the input and target sequences as tensors.

7. Defining the Bigram Language Model

```
class BigramLanguageModel(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)
```

- **class BigramLanguageModel(nn.Module)::** Defines a class for the bigram language model, inheriting from **nn.Module**, which is the base class for all neural network models in PyTorch.

- **def __init__(self, vocab_size):** The constructor method initializes the model. It takes `vocab_size` as an argument to define the size of the embedding layer.
- **super().__init__():** Calls the `__init__` method of the parent class `nn.Module`.
- **self.token_embedding_table = nn.Embedding(vocab_size, vocab_size):** Creates an embedding table where both the input and output dimensions are equal to the vocabulary size. This allows the model to learn a mapping from tokens to token probabilities.

```
def forward(self, index, targets=None):
    logits = self.token_embedding_table(index)
```

- **def forward(self, index, targets=None):** The forward method takes `index` (input sequence) and `targets` (target sequence) as arguments. This

is where the actual computations of the model occur.

- **logits = self.token_embedding_table(index):** Passes the input `index` through the embedding layer to produce `logits`, which are raw predictions (unnormalized probabilities).

if targets is None:

```
    loss = None
```

else:

```
    B, T, C = logits.shape
```

```
    logits = logits.view(B * T, C)
```

```
    targets = targets.view(B * T)
```

```
    loss = F.cross_entropy(logits, targets)
```

- **if targets is None:** If no `targets` are provided, the function doesn't compute a loss (this happens during inference).
- **else:** If `targets` are provided, the loss is calculated during training.
- **B, T, C = logits.shape:** This unpacks the shape of `logits` into `B` (batch size), `T` (sequence length), and `C` (vocabulary size).
- **logits = logits.view(B * T, C):** Reshapes `logits` to a 2D tensor of shape `(B * T, C)` for compatibility with the loss function.
- **targets = targets.view(B * T):** Reshapes `targets` to a 1D tensor of shape `(B * T)`.
- **loss = F.cross_entropy(logits, targets):** Computes the cross-entropy loss between the logits and the targets. This measures how well the model's predictions match the actual targets.

8. Token Generation

```
def generate(self, index, max_new_tokens):
    for _ in range(max_new_tokens):
        logits, loss = self.forward(index)
        logits = logits[:, -1, :] # focus only on the last time step
        probs = F.softmax(logits, dim=-1) # (B, C)
        index_next = torch.multinomial(probs, num_samples=1) # (B, 1)
        index = torch.cat((index, index_next), dim=1) # (B, T+1)
    return index
```

- **def generate(self, index, max_new_tokens):**: The **generate** function generates new tokens (characters) given an initial input **index**. It generates **max_new_tokens** new tokens.
- **for _ in range(max_new_tokens):**: Loops for **max_new_tokens** iterations, generating one token at a time.
- **logits, loss = self.forward(index)**: Performs a forward pass on the current input **index** to get the logits (predictions).
- **logits = logits[:, -1, :]**: Selects the logits from the last time step (since it's autoregressive, we only care about the prediction for the next token).
- **probs = F.softmax(logits, dim=-1)**: Converts logits into probabilities using the softmax function along the last dimension (vocabulary dimension).
- **index_next = torch.multinomial(probs, num_samples=1)**: Samples one token from the probability distribution using the **multinomial** function.
- **index = torch.cat((index, index_next), dim=1)**: Appends the generated token to the input sequence **index**.
- **return index**: Returns the final sequence with the generated tokens.

9. Model Initialization

```
model = BigramLanguageModel(vocab_size)
m = model.to(device)
```

- **model = BigramLanguageModel(vocab_size)**: Initializes the bigram language model with the **vocab_size**.
- **m = model.to(device)**: Moves the model to the chosen device (**cuda** or **cpu**).

10. Generating Text

```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
generated_chars = decode(m.generate(context, max_new_tokens=500)[0].tolist())
print(generated_chars)
```

- `context = torch.zeros((1, 1), dtype=torch.long, device=device)`: Initializes a tensor `context` with a shape of `(1, 1)` and fills it with zeros. This represents the initial state of the model.
- `generated_chars = decode(m.generate(context, max_new_tokens=500)[0].tolist())`: Calls the `generate` function to generate 500 new tokens. The generated token indices are decoded back into characters using the `decode` function.
- `print(generated_chars)`: Prints the generated text.

11. Training Loop

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
```

- `optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)`: Initializes the AdamW optimizer with the model parameters and the specified learning rate.

```
for iter in range(max_iters):
```

```
    if iter % eval_iters == 0:
```

```
        losses = estimate_loss()
```

```
        print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")
```

- `for iter in range(max_iters)::` The training loop runs for `max_iters` iterations.
- `if iter % eval_iters == 0::` Every `eval_iters` iterations, the model is evaluated on the training and validation sets.
- `losses = estimate_loss()`: Calls a function to estimate the loss (not defined in the provided code).
- `print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")`: Prints the current iteration and the corresponding losses.

```
xb, yb = get_batch('train')
```

```
logits, loss = model.forward(xb, yb)
```

```
optimizer.zero_grad(set_to_none=True)
```

```
loss.backward()
```

```
optimizer.step()
```

- `xb, yb = get_batch('train')`: Gets a batch of training data.

- `logits, loss = model.forward(xb, yb)`: Performs a forward pass on the training batch and computes the loss.
- `optimizer.zero_grad(set_to_none=True)`: Clears the gradients of all optimized tensors. The `set_to_none=True` optimizes performance.
- `loss.backward()`: Computes the gradients of the loss with respect to the model parameters.
- `optimizer.step()`: Updates the model parameters using the computed gradients.

12. Loss and Output

```
print(loss.item())
```

- `print(loss.item())`: Prints the current loss value (the `item()` method is used to extract the scalar value from the loss tensor).

UI IMPLEMENTATION

End-to-End Project Overview: User Interaction with a Language Model

1. Introduction: This project demonstrates a simple interactive text generation model using a pre-trained GPT-style language model. The user provides a text prompt, and the model generates a continuation of the text, simulating natural language completion. This implementation can be used as an AI-based text generation system, which can be applied to various domains like content creation, chatbots, and more.

2. User Interface (UI): The user interface (UI) is minimal and text-based, providing an interactive command-line prompt where users can input a sentence or partial text. The system then responds with generated content based on the provided input. Below are the steps involved in the UI flow:

Detailed Breakdown of the UI Flow:

Step 1: Command-Line Argument Parsing

The UI begins by accepting a batch size for the model via command-line arguments. The user must provide this argument when executing the script. This is done using Python's `argparse` library, which handles command-line inputs in a structured manner.

```
parser = argparse.ArgumentParser(description='This is a demonstration program')
```

```
parser.add_argument('-batch_size', type=str, required=True, help='Please provide a batch_size')
args = parser.parse_args()
```

- **Purpose:** The batch size controls how many inputs the model processes at once. The value is provided when starting the script from the terminal (e.g., `python script.py -batch_size 16`).
- **User Action:** The user must specify the batch size when running the program.

Step 2: Display the Batch Size and Set Device

Once the argument is parsed, the `batch_size` is printed, and the system checks if CUDA (GPU) is available. If CUDA is present, the model will run on the GPU; otherwise, it defaults to the CPU.

```
print(f'batch size: {args.batch_size}')
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

- **Purpose:** Displays the batch size and sets the computational device (GPU or CPU).
- **User Action:** The user is informed about the batch size and the chosen computational device.

Step 3: Load the Model

The model, which is a GPT-style language model, is loaded from a pre-trained state saved in a file (`model-01.pkl`). This step loads the model parameters from the disk.

```
print('loading model parameters...')
with open('model-01.pkl', 'rb') as f:
    model = pickle.load(f)
print('loaded successfully!')
```

- **Purpose:** The model is loaded into memory, ready to generate text.
- **User Action:** The user waits for the model to load successfully before interacting with it.

Step 4: Prompt Input

The program then enters an infinite loop where it waits for the user to input a text prompt. This is the core of the user interaction, where the user provides text to guide the generation process.

```
while True:
    prompt = input("Prompt:\n")
```

- **Purpose:** This waits for the user to type a prompt and hit Enter.
- **User Action:** The user types a text prompt (e.g., "Once upon a time, in a land far away").

Step 5: Text Encoding

After the user submits a prompt, the text is encoded into integer indices. This encoding allows the model to process the input, as neural networks generally work with numerical data. The encoding uses the `string_to_int` dictionary, where each character in the input is mapped to a corresponding integer.

```
context = torch.tensor(encode(prompt), dtype=torch.long, device=device)
```

- **Purpose:** Converts the user's text into a format the model can understand (tensor of indices).
- **User Action:** No direct action from the user, but they should expect that the model can understand their input.

Step 6: Text Generation

Once the input text is encoded, the model uses its `generate` function to predict the next characters based on the provided prompt. This is done using the model's forward pass, where it processes the encoded input, computes probabilities, and samples the next token.

```
generated_chars = decode(m.generate(context.unsqueeze(0),
max_new_tokens=150)[0].tolist())
```

- **Purpose:** This generates the continuation of the text by sampling tokens one by one.
- **User Action:** The user waits as the model generates the next sequence of characters based on their input.

Step 7: Display the Output

Once the model generates the continuation of the text, it is decoded back into human-readable characters using the `decode` function. The generated text is then printed to the screen as the model's response.

```
print(f'Completion:\n{generated_chars}')
```

- **Purpose:** Displays the model's generated text to the user.
- **User Action:** The user sees the model's completion, which could be something like "Once upon a time, in a land far away, there was a magical forest."

End-to-End Interaction Example:

Here's what the full interaction might look like when running the program from the command line:

Starting the Program:

```
$ python script.py -batch_size 16  
batch size: 16  
loading model parameters...  
loaded successfully!
```

1.

User Input:

Prompt:
Once upon a time, in a land far away,

2.

Model Output:

Completion:
there was a magical forest filled with creatures of every kind. The trees whispered ancient secrets, and the sky was always painted in shades of purple and gold.

3.

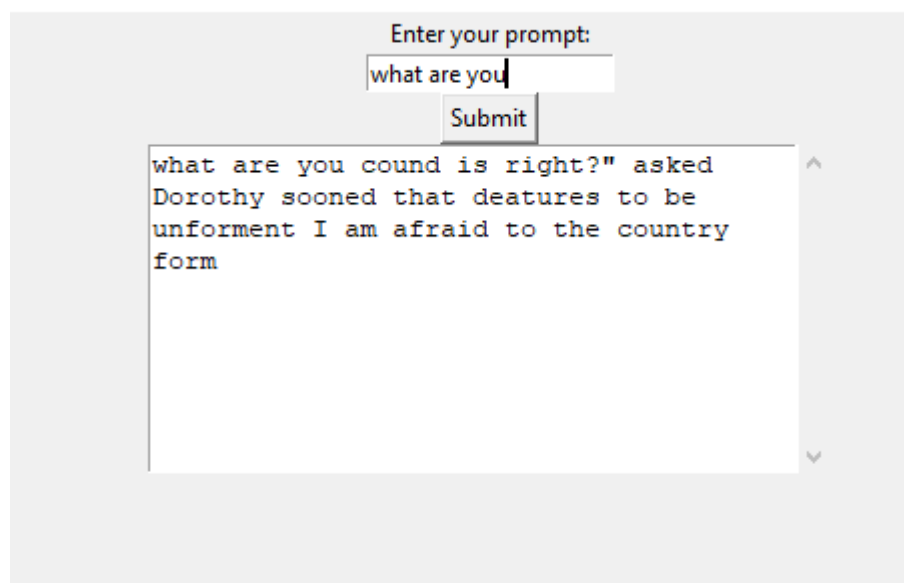
Key Features of the UI Implementation:

1. **Simple and Minimalist:** The interface is entirely text-based, with the user providing prompts and receiving completions via the command line. This keeps the interaction simple and focused on the core functionality of text generation.
2. **Real-Time Interaction:** The system is designed to respond to user input in real-time. The user can enter a prompt and immediately see the model's response.
3. **Flexibility:** The system allows the user to enter any prompt, and the model will generate a continuation based on the given input. This makes it versatile for a wide range of tasks,

from creative writing to idea generation.

4. **Model-Driven Output:** The generated text is based on the model's learned knowledge, enabling the system to provide human-like text completions. It takes into account the prompt entered by the user and generates coherent, contextually relevant continuations.

RESULTS



The image shows a web-based interface for text generation. At the top, there is a label "Enter your prompt:" followed by a text input field containing the text "what are you". To the right of the input field is a "Submit" button. Below the input field, the generated text is displayed in a white box with a light gray border. The text is: "what are you cound is right?" asked Dorothy sooned that deatures to be unforment I am afraid to the country form". The text is formatted with a monospaced font and includes a line break. There are small upward and downward arrow icons on the right side of the output box.

Enter your prompt:

what are you

Submit

what are you cound is right?" asked
Dorothy sooned that deatures to be
unforment I am afraid to the country
form

