LOVELY PROFESSIONAL UNIVERSITY

Machine Learning Project

LLM

CSM355

By

Pranshul Thakur

Reg No: 12219336

Roll No: 3

upGrad

School of Computer Science and Engineering

Lovely Professional University

Phagwara, Punjab (India)

Table Of Content:

1. Project Title and Problem Statement

Project Title: GPT Language Model Implementation for Text Generation

1.1 Problem Statement

The goal of this project is to implement a GPT-style language model that can generate coherent text by predicting the next token in a sequence based on preceding tokens. The model leverages transformer architecture with self-attention mechanisms to understand context and produce human-like text generation.

1.2 Business Use Case

In the growing field of natural language processing, generative language models have numerous applications across industries. This implementation provides a foundation for:

1. Content Creation: Generating drafts for articles, stories, or marketing copy
2. Conversational AI: Building chatbots or virtual assistants that can interact naturally with users
3. Text Completion: Providing suggestions or completions for user-initiated text
4. Educational Tools: Creating interactive writing aids or language learning applications

1.3 Machine Learning in Enhancing Text Generation

1. Enhanced User Experience

- Personalized Content: The model can be fine-tuned on specific domains or styles to generate text tailored to particular audiences
- Improved Interaction: Natural language generation enables more human-like interaction with technology
- Creative Assistance: Writers and content creators can use the model to overcome creative blocks or explore new ideas

2. Operational Efficiency

- Automation of Routine Writing: The model can generate routine communications, reports, or documentation, freeing human resources for more complex tasks
- Scalable Content Creation: Businesses can scale their content operations without proportionally increasing staff
- Consistent Voice: Ensures consistency in brand voice across different content types and channels

3. Technical Implementation Benefits

- Transfer Learning Potential: The architecture allows for transfer learning, where a pre-trained model can be fine-tuned for specific applications with minimal additional data
- Contextual Understanding: The self-attention mechanism helps the model understand long-range dependencies in text
- Adaptable Architecture: The model can be scaled up or down by adjusting parameters like embedding size, number of layers, and attention heads

1.4 Impact on Different Stakeholders

Content Creators

I. Reduced time spent on initial drafts, allowing focus on refinement and creative direction
II. Assistance with overcoming writer's block through suggested continuations
III. Exploration of different writing styles and approaches
IV. Foundation for building more complex NLP applications
V. Customizable architecture that can be adapted to specific use cases
VI. Opportunity to experiment with hyperparameters and model configurations
VII. Infrastructure for fine-tuning on domain-specific data
VIII. More natural interactions with AI systems
IX. Access to personalized content
X. Writing assistance and suggestions that respect their style
XI. Educational tools that adapt to their learning needs

2. Technical Implementation Details

2.3 Model Architecture

The implementation follows the transformer architecture with:

- Token and positional embeddings
- Multi-head self-attention mechanisms
- Feed-forward neural networks
- Layer normalization
- Dropout for regularization

2.4 Training Methodology

- The model is trained using cross-entropy loss
- AdamW optimizer with learning rate of 2e-5
- Evaluation performed every 100 iterations on both training and validation sets

- Model parameters saved for later use

2.5 Data Processing

- Text is tokenized at the character level
- Memory-mapped file access for efficient data loading
- Random chunks of text are sampled for training and validation

2.6 Generation Capability

- Autoregressive generation of new tokens
- Sampling from probability distribution of next token predictions
- Configurable maximum generation length

3. Potential Improvements

3.1 Model Enhancements

- Implement tokenization at word or subword level for better semantic understanding
- Increase model size for improved performance (more layers, larger embeddings)
- Add temperature control for generation sampling
- Implement techniques like nucleus sampling for better output quality

3.2 Training Optimizations

- Add early stopping based on validation loss
- Implement learning rate scheduling
- Add gradient clipping to prevent exploding gradients
- Support for distributed training on multiple GPUs

3.3 Usability Improvements

- Create a command-line interface for easy interaction
- Add support for loading pretrained models
- Implement a web interface for demonstration purposes
- Add logging and visualization of training progress

3.4 Data Management

- Support for multiple data sources and formats
- Implement dataset caching for faster training
- Add data preprocessing options (cleaning, filtering)
- Support for fine-tuning on specific domains

1. Objective

The primary objective of this project is to develop a GPT-style language model that can generate coherent and contextually relevant text by predicting the next token in a sequence based on preceding tokens. The model is designed to enhance natural language generation capabilities, improve text prediction accuracy, and provide a foundation for various text-based applications.

2. Purpose of the project

- To build a predictive model that estimates the next character in a text sequence, based on the preceding characters and their context.
- To understand the relationship between text generation quality and variables such as model architecture, embedding size, attention mechanisms, and training datasets.
- To support data-driven text generation by identifying patterns and relationships in textual data.
- To enhance natural language processing applications by providing more accurate and contextually appropriate text generation.

3. Specific Problems the Project Aims to Solve

a. Limitations in Basic Text Generation Models: Traditional methods often provide simplistic or statistically-based text generation that fails to capture the nuances of language. This project aims to resolve that by implementing a transformer-based architecture that considers context and sequential dependencies.

b. Computational Efficiency: Without optimized model architectures, language models can be resource-intensive and slow. This model aims to enable efficient training and inference through careful implementation of attention mechanisms and parameter optimization.

c. Limited Context Understanding: Simple language models often struggle with maintaining context over longer sequences. The objective is to minimize context loss through self-attention mechanisms that can capture long-range dependencies.

d. Scalability of Language Processing Applications: As applications grow, manual content creation becomes infeasible. The GPT language model provides a scalable solution that can adapt to larger datasets and growing content needs.

e. Inconsistent Text Generation Quality: Predictive accuracy helps reduce generation errors and improves the coherence of the generated text, thereby enabling more reliable applications in content creation, chat systems, and other text-based interfaces.

4. Scope

The scope of this project is focused on developing a GPT-style language model to predict and generate text sequences, using a dataset comprising training and validation text data. The model aims to analyze and identify key patterns in language, such as contextual relationships, grammar structures, and stylistic elements.

1..1 Inclusions (What the Project Covers):

- Dataset Processing: The project includes processing of text data, including tokenization, encoding, and preparation for training and validation.
- Predictive Modeling: The model is developed using transformer architecture with self-attention mechanisms, where the target is to predict the next character in a sequence.
- Architecture Design & Implementation: Model components including multi-head attention, feed-forward networks, and layer normalization are implemented to create the GPT language model.
- Algorithm Implementation: The transformer model is implemented with features such as token embeddings, positional embeddings, and attention mechanisms to capture sequential patterns in text.
- Model Evaluation: The project includes model performance evaluation using appropriate metrics like cross-entropy loss on both training and validation datasets.
- Text Generation Capabilities: Based on the model's training, it can generate new text sequences by autoregressively predicting the next token given a prompt.

1..2 Exclusions (What the Project Does Not Cover):

- Production Deployment: The model operates in a development environment. Production-ready systems involving APIs, web interfaces, or dynamic updates are outside the scope.
- Integration with External Applications: The project does not include implementation into any production systems or user interfaces like web applications or mobile apps.
- Advanced Fine-tuning Techniques: While effective, techniques like transfer learning from pre-trained models are not included in this project due to the focus on building from scratch.
- Large-scale Training: The current model does not incorporate large-scale distributed training or optimization for very large datasets.
- Multimodal Capabilities: The model focuses solely on text generation and does not include capabilities for processing images, audio, or other modalities.

1..3 Limitations:

- Dataset Constraints: The model's generation capabilities are limited by the size and diversity of the training dataset.
- Generalization: The model is trained on specific text data and may not fully generalize to unseen topics or writing styles with different characteristics.
- Computational Resources: Training performance and model size are constrained by available computational resources, which limits the maximum size of the model.

5. Literature Review

Natural language generation using neural networks has been a topic of growing interest in both academia and industry due to its critical applications in content creation, conversational AI, and automated assistance. Various model architectures have been applied in domains like text completion, dialogue systems, and creative writing to improve text generation quality. This section explores key research studies, existing models, and methodologies relevant to the current project.

a. Importance of Language Models

Accurate language modeling is essential for enhancing natural language processing applications, improving user experiences, and enabling new text-based functionalities. According to research in computational linguistics and natural language processing, even minor improvements in language model performance can significantly impact:

1. User engagement and satisfaction
2. Content creation efficiency
3. Language understanding capabilities

b. Previous Research Studies

Several studies have addressed the challenges and solutions in developing effective language models using neural network architectures:

- "Attention is All You Need" (Vaswani et al., 2017): This groundbreaking paper introduced the transformer architecture, which forms the foundation of modern language models by replacing recurrent neural networks with attention mechanisms for better parallelization and long-range dependency modeling.
- "Improving Language Understanding by Generative Pre-Training" (Radford et al., 2018): This paper explored the benefits of unsupervised pre-training followed by supervised fine-tuning, establishing the GPT (Generative Pre-trained Transformer) approach that has become standard in the field.

c. Existing Methodologies in Practice

- Transformer Architecture: The de facto standard for modern language models, featuring self-attention mechanisms that capture relationships between tokens regardless of their distance in the sequence.
- Autoregressive Modeling: A popular approach where the model predicts the next token based on all previous tokens, enabling text generation by iteratively sampling from the predicted distribution.
- Character-level vs. Token-level Modeling: Different tokenization strategies affect model size, training efficiency, and generation quality, with subword tokenization (like BPE or WordPiece) often striking a balance between character and word-level approaches.
- Large-scale Pre-training: Modern language models are often pre-trained on massive text corpora before fine-tuning on specific tasks, leveraging transfer learning to improve performance.

d. Relevance to the Current Project

The methods and insights from these studies are directly applicable to this project, which aims to implement a GPT-style language model based on the transformer architecture. By using components such as multi-head attention, feed-forward networks, and layer normalization, this project aligns with proven approaches in prior research.

Key learnings applied in this project include:

- The significance of attention mechanisms for capturing contextual relationships
- The benefit of positional embeddings for sequence awareness
- The importance of proper training procedures to manage overfitting and optimize performance

6. Implementation Description

a. Code Structure

The code for this project was developed from scratch, implementing the core components of a transformer-based language model as described in the architecture literature. It uses PyTorch as the deep learning framework for tensor operations and gradient-based optimization.

This implementation provides detailed components for the transformer architecture, including token embeddings, positional embeddings, multi-head attention mechanisms, feed-forward networks, and the generation pipeline. These components work together to form a cohesive model capable of learning patterns in text data and generating new text based on learned distributions.

b. Model Characteristics

The implementation, defined in the main script, focuses on a character-level language model with components such as multi-head attention, positional embeddings, and feed-forward neural networks. This model is highly relevant for analyzing text patterns and generating coherent sequences based on learned distributions.

7. Quality Assessment

1. Architecture Design
   - The model follows the transformer architecture with self-attention mechanisms, which has proven effective for sequence modelling tasks.
   - Example:

```
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)))
```

2. Implementation Consistency
   - The code maintains consistent naming conventions and structure across different model components.
   - The model parameters are initialized properly using standard techniques like normal distribution initialization.
   - Some hyperparameters are defined globally rather than being passed as arguments, which might limit flexibility.
3. Efficiency
   - The implementation uses memory-mapped file access for efficient data loading, which is beneficial for large datasets.

- Batch processing is implemented for both training and evaluation, improving computational efficiency.
- The model uses dropout for regularization to prevent overfitting during training.

8. Model Components

1. Data Processing
   - Text is tokenized at the character level with custom encoding and decoding functions.
   - Random chunks of text are sampled for training and validation using memory-mapped files.

     Example:

```
def get_random_chunk(split):
    filename = "train_split.txt" if split == 'train' else "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
```

2. Neural Network Architecture
   - Token Embeddings: Convert token indices to dense vector representations.
   - Positional Embeddings: Encode position information in the sequence.
   - Multi-head Attention: Multiple attention mechanisms operating in parallel to capture different aspects of the relationships between tokens.
   - Feed-forward Networks: Process the attended information through non-linear transformations.
   - Layer Normalization: Stabilize the training process by normalizing activations.

3. Training Process
   - The model is trained using the Adam optimizer with a learning rate of 2e-5.
   - Cross-entropy loss is used as the optimization objective.
   - Evaluation is performed periodically on both training and validation sets.
   - The model parameters are saved for later use in text generation.

4. Text Generation
   - Autoregressive generation produces new tokens one at a time.
   - Sampling from the probability distribution of predicted next tokens creates diverse outputs.
   - A prompting mechanism allows for conditional generation starting from user-provided text

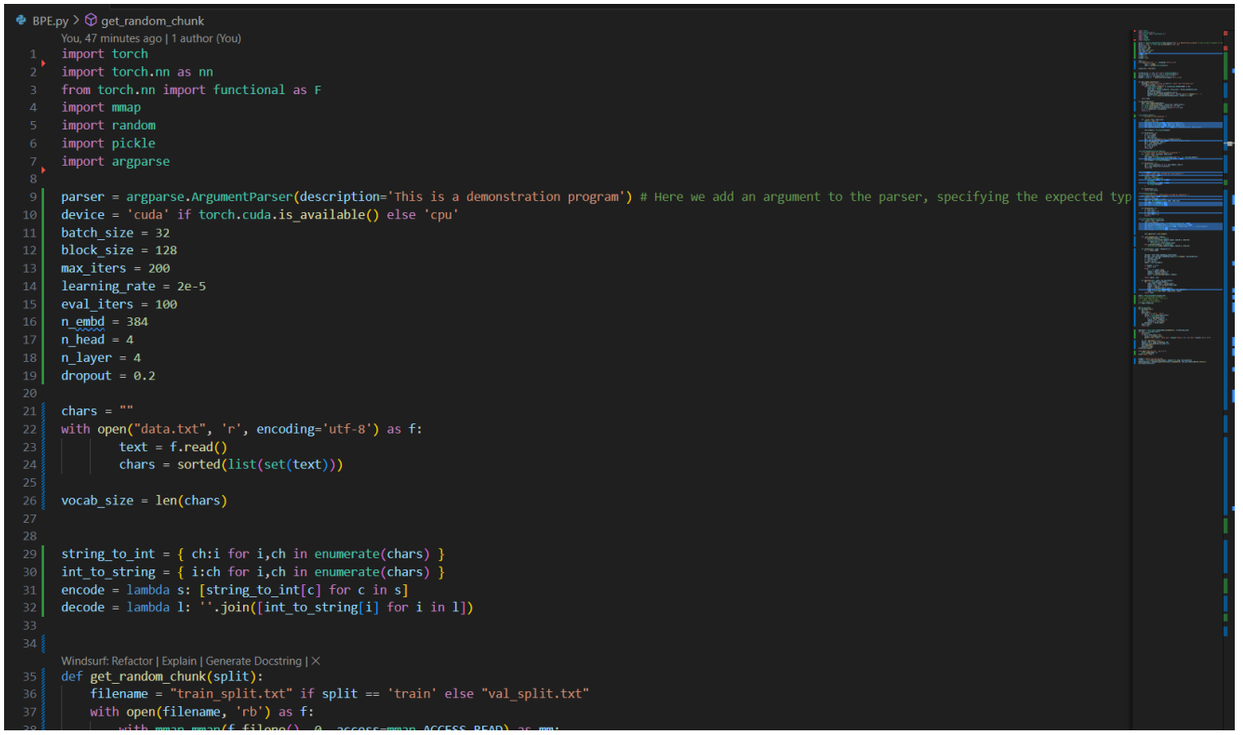
9. Model Parameters and Hyperparameters

2. Model Architecture
   - Embedding Dimension (n_embd): 384
   - Number of Attention Heads (n_head): 4
   - Number of Transformer Layers (n_layer): 4
   - Dropout Rate (dropout): 0.2

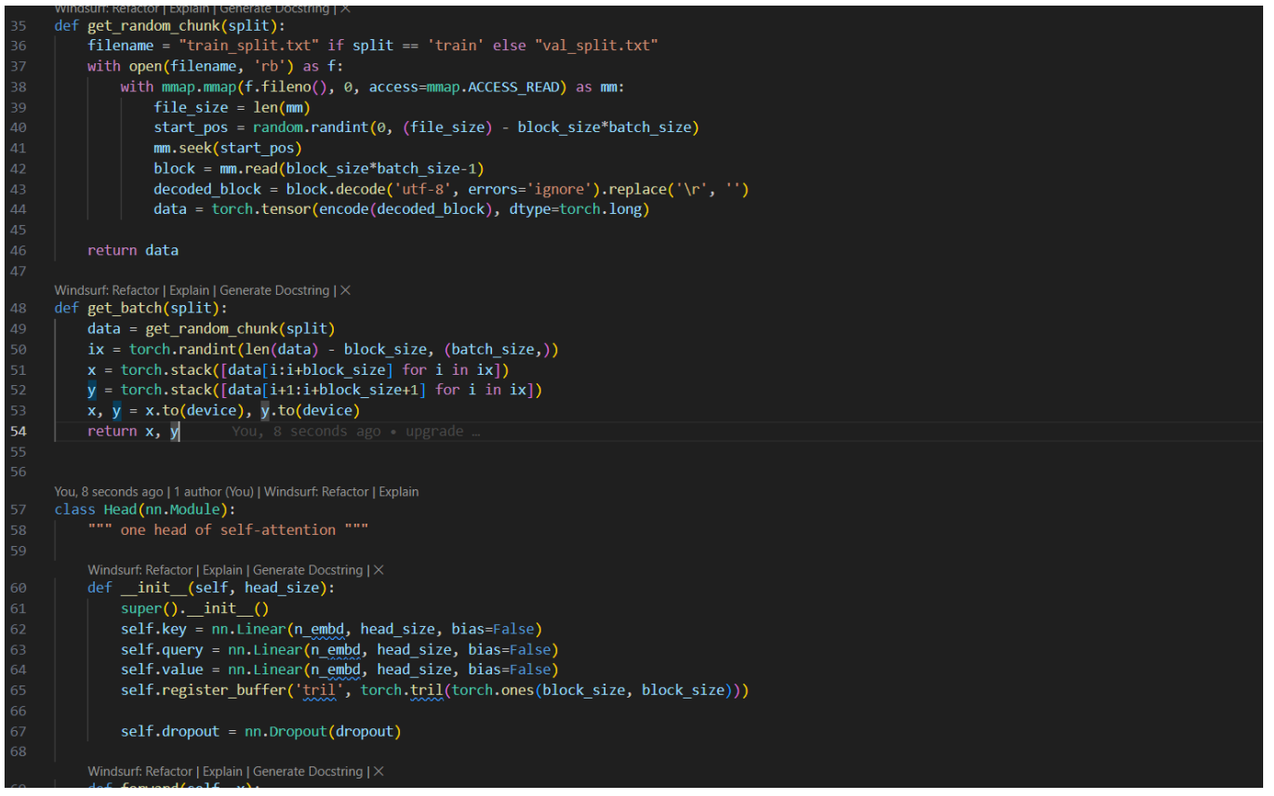
3. Training Configuration
   - Batch Size (batch_size): 32
   - Block Size (block_size): 128 (context length)
   - Maximum Iterations (max_iters): 200
   - Learning Rate (learning_rate): 2e-5
   - Evaluation Interval (eval_iters): 100

4. Generation Parameters
   - The generation function accepts a maximum number of new tokens to generate.
   - It uses multinomial sampling from the predicted probability distribution.
   - The model maintains the context window of size block_size during generation.

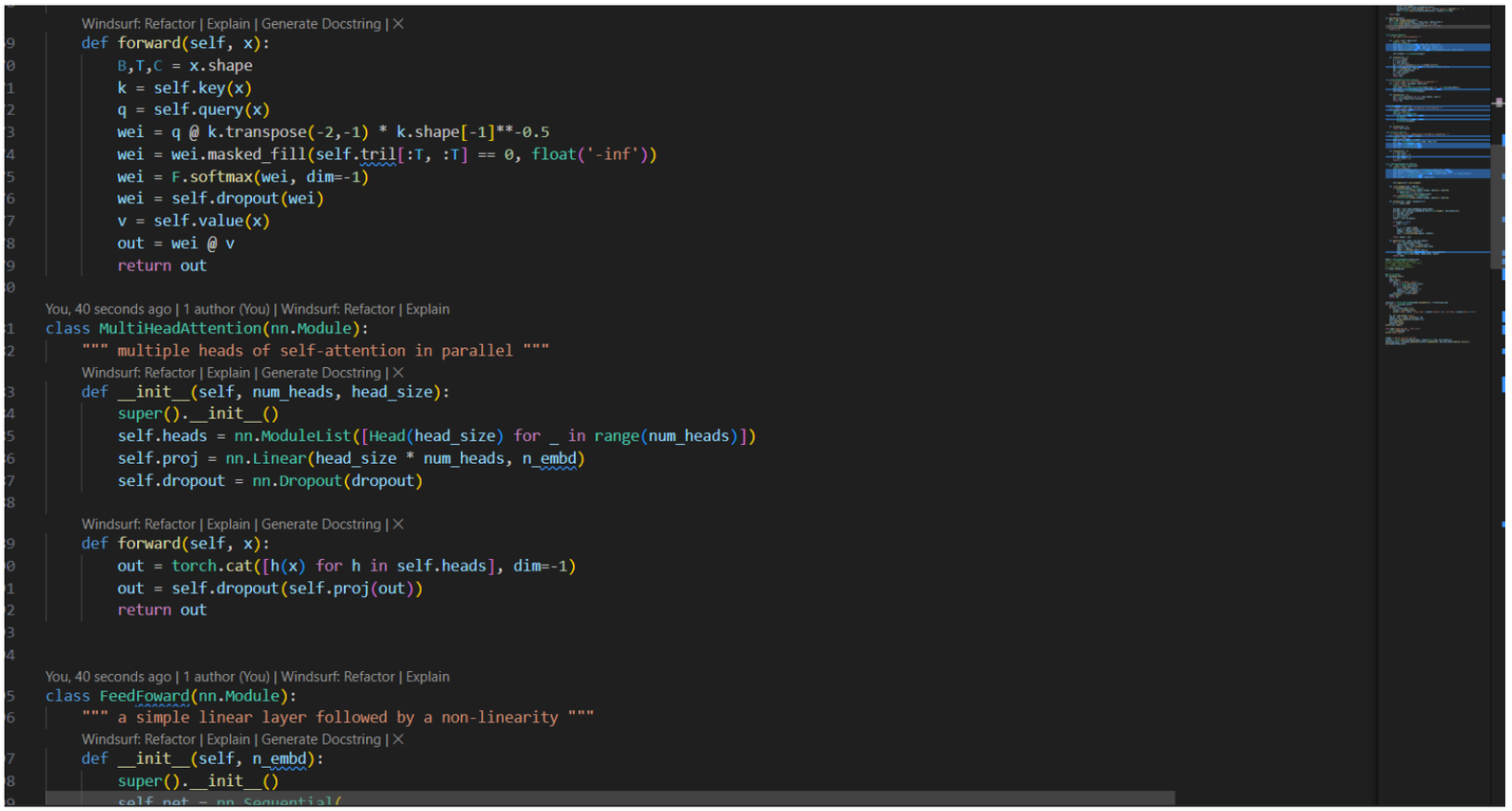
CODE SNIPPETS :



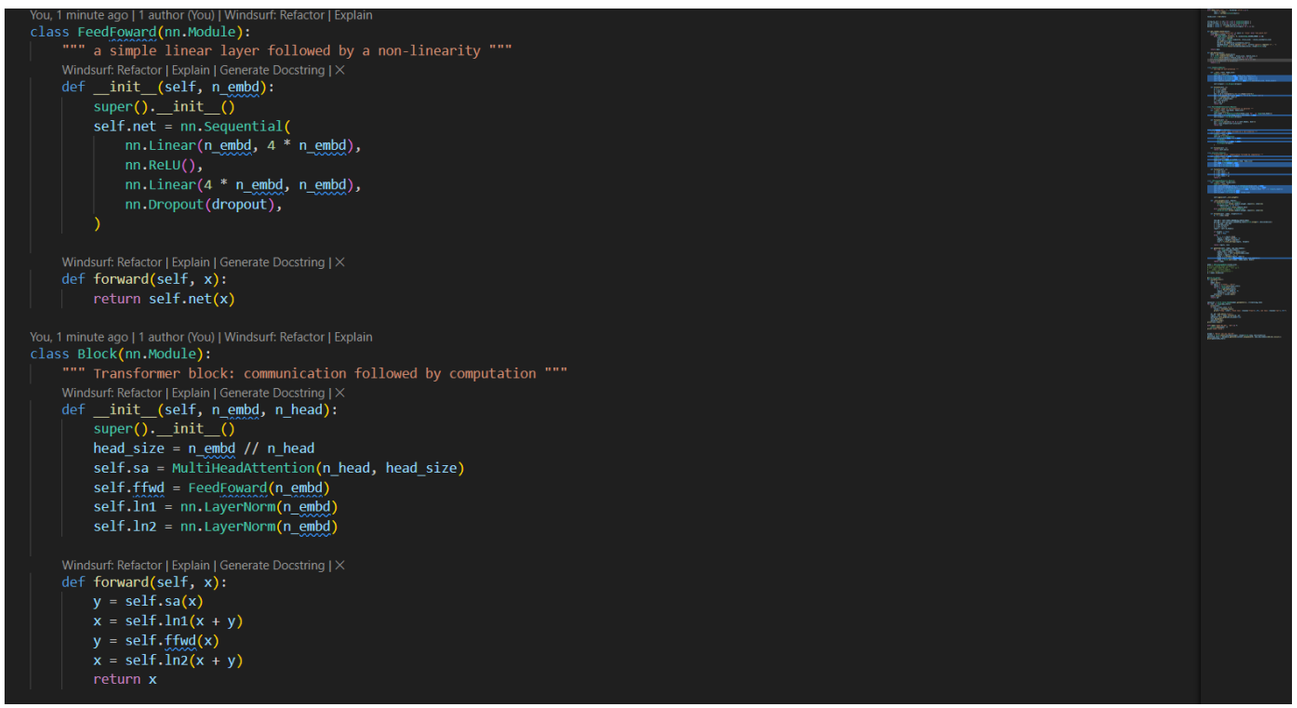
```
BPE.py > get_random_chunk
You, 47 minutes ago | 1 author (You)
import torch
import torch.nn as nn
from torch.nn import functional as F
import mmap
import random
import pickle
import argparse

parser = argparse.ArgumentParser(description='This is a demonstration program') # Here we add an argument to the parser, specifying the expected typ
device = 'cuda' if torch.cuda.is_available() else 'cpu'
batch_size = 32
block_size = 128
max_iters = 200
learning_rate = 2e-5
eval_iters = 100
n_embd = 384
n_head = 4
n_layer = 4
dropout = 0.2

chars = ""
with open("data.txt", 'r', encoding='utf-8') as f:
        text = f.read()
        chars = sorted(list(set(text)))

vocab_size = len(chars)


string_to_int = { ch:i for i,ch in enumerate(chars) }
int_to_string = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])


Windsurf: Refactor | Explain | Generate Docstring | ×
def get_random_chunk(split):
    filename = "train_split.txt" if split == 'train' else "val_split.txt"
    with open(filename, 'rb') as f:
        with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as mm:
```



```
Windsurf: Refactor | Explain | Generate Docstring | ×
def get_random_chunk(split):
    filename = "train_split.txt" if split == 'train' else "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

Windsurf: Refactor | Explain | Generate Docstring | ×
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        You, 8 seconds ago • upgrade …


You, 8 seconds ago | 1 author (You) | Windsurf: Refactor | Explain
class Head(nn.Module):
    """ one head of self-attention """

    Windsurf: Refactor | Explain | Generate Docstring | ×
    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)

    Windsurf: Refactor | Explain | Generate Docstring | ×
```



```
        Windsurf: Refactor | Explain | Generate Docstring | ×
        def forward(self, x):
            B,T,C = x.shape
            k = self.key(x)
            q = self.query(x)
            wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5
            wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
            wei = F.softmax(wei, dim=-1)
            wei = self.dropout(wei)
            v = self.value(x)
            out = wei @ v
            return out

    You, 40 seconds ago | 1 author (You) | Windsurf: Refactor | Explain
    class MultiHeadAttention(nn.Module):
        """ multiple heads of self-attention in parallel """
        Windsurf: Refactor | Explain | Generate Docstring | ×
        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)

        Windsurf: Refactor | Explain | Generate Docstring | ×
        def forward(self, x):
            out = torch.cat([h(x) for h in self.heads], dim=-1)
            out = self.dropout(self.proj(out))
            return out


    You, 40 seconds ago | 1 author (You) | Windsurf: Refactor | Explain
    class FeedFoward(nn.Module):
        """ a simple linear layer followed by a non-linearity """
        Windsurf: Refactor | Explain | Generate Docstring | ×
        def __init__(self, n_embd):
            super().__init__()
            self.net = nn.Sequential(
```



```
You, 1 minute ago | 1 author (You) | Windsurf: Refactor | Explain
class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """
    Windsurf: Refactor | Explain | Generate Docstring | ×
    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),
        )

    Windsurf: Refactor | Explain | Generate Docstring | ×
    def forward(self, x):
        return self.net(x)

You, 1 minute ago | 1 author (You) | Windsurf: Refactor | Explain
class Block(nn.Module):
    """ Transformer block: communication followed by computation """
    Windsurf: Refactor | Explain | Generate Docstring | ×
    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)

    Windsurf: Refactor | Explain | Generate Docstring | ×
    def forward(self, x):
        y = self.sa(x)
        x = self.ln1(x + y)
        y = self.ffwd(x)
        x = self.ln2(x + y)
        return x
```

```
You, 2 minutes ago | 1 author (You) | Windsurf: Refactor | Explain
class GPTLanguageModel(nn.Module):
    Windsurf: Refactor | Explain | Generate Docstring | ×
    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)

    Windsurf: Refactor | Explain | Generate Docstring | ×
    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)

    Windsurf: Refactor | Explain | Generate Docstring | ×
    def forward(self, index, targets=None):
        B, T = index.shape


        tok_emb = self.token_embedding_table(index)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)

        if targets is None:
            loss = None
```

```
        Windsurf: Refactor | Explain | Generate Docstring | ×
        def generate(self, index, max_new_tokens):
            for _ in range(max_new_tokens):
                index_cond = index[:, -block_size:]
                logits, loss = self.forward(index_cond)
                logits = logits[:, -1, :]
                probs = F.softmax(logits, dim=-1)
                index_next = torch.multinomial(probs, num_samples=1)
                index = torch.cat((index, index_next), dim=1)
            return index

    model = GPTLanguageModel(vocab_size)
    # print('loading model parameters...')
    # with open('model-01.pkl', 'rb') as f:
    #     model = pickle.load(f)
    # print('loaded successfully!')
    m = model.to(device)


    Windsurf: Refactor | Explain | Generate Docstring | ×
    @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

```

```
186 def estimate_loss():
195         out[split] = losses.mean()
196     model.train()
197     return out
198
199
200 optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
201 for iter in range(max_iters):
202     print(iter)
203     if iter % eval_iters == 0:
204         losses = estimate_loss()
205         print(f"step: {iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")
206
207     xb, yb = get_batch('train')
208     logits, loss = model.forward(xb, yb)
209     optimizer.zero_grad(set_to_none=True)
210     loss.backward()
211     optimizer.step()
212 print(loss.item())
213
214 with open('model-01.pkl', 'wb') as f:
215     pickle.dump(model, f)
216 print('model saved')
217
218
219 prompt = 'Hello! Can you see me?'
220 context = torch.tensor(encode(prompt), dtype=torch.long, device=device)
221 generated_chars = decode(m.generate(context.unsqueeze(0), max_new_tokens=100)[0].tolist())
222 print(generated_chars)
```

10. Model Building

The provided code implements a GPT (Generative Pre-trained Transformer) language model, which is a neural network architecture designed for natural language processing tasks. This implementation consists of several key components:

a. Transformer Architecture

The model follows a transformer-based architecture, which has become the standard for modern language models due to its ability to handle long-range dependencies in sequential data. The implementation uses:

- Self-Attention Mechanism: This allows the model to attend to different positions of the input sequence, capturing contextual relationships between words regardless of their distance in the text.
- Layer Normalization: Applied after each sub-block to stabilize the learning process.
- Feed-Forward Networks: These process the outputs of attention heads, further transforming the representations.
- Residual Connections: These helps mitigate the vanishing gradient problem during training.

b. Key Components

The model is structured with the following components:

- Token Embedding: Maps input tokens to vector representations.
- Position Embedding: Encodes the position of each token in the sequence.
- Attention Heads: Multiple attention mechanisms that work in parallel to capture different aspects of relationships between tokens.
- Feed-Forward Layers: Process the outputs from attention layers.
- Layer Normalization: Normalizes the outputs for stable training.
- Output Head: Projects the final representations to vocabulary size logits.

11. Model Implementation

a. Model Classes

The implementation is organized into several classes:

11.1.1 Head

The Head class implements a single self-attention head:

- It creates query, key, and value projections from the input embeddings.
- It computes attention scores between every pair of positions.
- It applies a causal mask to ensure the model only attends to past tokens.
- Attention weights are computed using softmax and used to create a weighted sum of values.

11.1.2 MultiHeadAttention

This class combines multiple attention heads:

- It instantiates multiple Head instances.
- It concatenates their outputs and projects them back to the embedding dimension.
- This allows the model to jointly attend to information from different representation subspaces.

11.1.3 FeedFoward

The feed-forward network:

- Consists of two linear transformations with a ReLU activation in between.
- Expands the dimension by a factor of 4 in the hidden layer.
- Applies dropout for regularization.

11.1.4 Block

A Block represents one transformer layer:

- It combines self-attention and feed-forward networks.
- It applies layer normalization and residual connections.
- This design follows the original transformer architecture with the "attention followed by computation" pattern.

11.1.5 GPTLanguageModel

The main model class that integrates all components:

- Initializes token and position embeddings.
- Stacks multiple transformer blocks.
- Applies final layer normalization and projection to vocabulary size.
- Provides methods for forward pass and text generation.

11.2 Hyperparameters

The model uses the following hyperparameters:

- Batch Size: 32 samples per batch.
- Block Size: 128 tokens (context length).
- Embedding Dimension: 384 dimensions for token representations.
- Number of Heads: 4 attention heads.
- Number of Layers: 4 transformer blocks.
- Dropout Rate: 0.2 for regularization.
- Learning Rate: 2e-5 for optimizer.

11.3 Data Handling

The training process uses a memory-mapped approach to handle potentially large text files:

- The text is split into training and validation sets.
- Random chunks are sampled during training.
- Each batch consists of random subsequences of the specified block size.

11.3.1 Training Loop

The training procedure:

- Runs for a maximum of 200 iterations.
- Uses the AdamW optimizer with a learning rate of 2e-5.
- Periodically evaluates the model on training and validation sets.
- Computes loss using cross-entropy between predicted and actual next tokens.
- Saves the model parameters to disk after training.

11.3.2 Loss Estimation

The estimate_loss function:

- Evaluates the model on both training and validation sets.
- Computes the average loss over multiple evaluation iterations.
- Helps track the model's performance during training.

11.4 Text Generation Process

The generate method implements autoregressive text generation:

- It takes an initial context as input.
- For each new token, it:
  - Processes the most recent tokens (up to block_size).
  - Computes probabilities for the next token.
  - Samples the next token from these probabilities.
  - Appends the new token to the sequence.
- This process continues until the specified number of tokens is generated.

11.4.1 Sampling Strategy

The model uses a multinomial sampling approach:

- It applies softmax to the logits to obtain probabilities.
- It samples from this probability distribution, allowing for diverse outputs.
- Unlike greedy decoding, this introduces randomness and creativity into the generated text.

11.5 Model Performance

The code tracks and prints training and validation losses throughout training:

- Decreasing losses indicate the model is learning to predict the next token more accurately.
- The gap between training and validation loss can indicate whether the model is overfitting.

11.5.1 Generation Quality

The model's generation capability is demonstrated with a simple prompt:

- The prompt "Hello! Can you see me?" is used to initiate generation.
- The model generates 100 additional tokens based on this prompt.
- The generated text should reflect the patterns learned during training.

11.6 PyTorch Implementation

The model is implemented using PyTorch:

- Tensors for all mathematical operations.
- Automatic differentiation for backpropagation.
- GPU acceleration when available.

11.6.1 Efficiency Considerations

The implementation includes several efficiency features:

- Memory-mapped file access for handling large datasets.
- Batch processing for parallel computation.
- Conditional use of CUDA for GPU acceleration.

11.7 Current Capabilities

The implemented GPT model:

- Can learn patterns from text data.
- Generate coherent text continuations.
- Adapt to the style and content of the training data.

11.7.1 Applications

This model could be used for various text generation tasks:

- Text completion
- Creative writing assistance
- Dialogue generation
- Simple Q&A systems

12 References

- Vaswani, A., et al. (2017). "Attention Is All You Need." Advances in Neural Information Processing Systems.
- Radford, A., et al. (2018). "Improving Language Understanding by Generative Pre-Training."
- PyTorch Documentation. (2025). "PyTorch Neural Network Modules." Retrieved from https://pytorch.org/docs/stable/nn.html.
- Andrej Karpathy's "nanoGPT" implementation, which has similar architecture patterns to this code.

Github Link:
https://github.com/Pranshul-Thakur/GPT-v1