GPT-2 Fine-tuning Pipeline for Next Word Prediction:
A Comprehensive Code Analysis

Technical Documentation

July 9, 2025

Contents

1 Introduction

This document provides a comprehensive explanation of a GPT-2 fine-tuning pipeline designed for next word prediction tasks. The code implements a complete machine learning workflow from data preprocessing to model evaluation using PyTorch and Hugging Face Transformers.

1.1 Overview

The pipeline consists of several key components:

- Custom dataset handling for WikiText-2
- GPT-2 model fine-tuning using Hugging Face Transformers
- Comprehensive evaluation metrics (perplexity, top-k accuracy)
- Data exploration and visualization tools

1.2 Prerequisites

- Python 3.7+
- PyTorch
- Hugging Face Transformers
- Standard ML libraries (numpy, pandas, matplotlib)

2 Theoretical Background

2.1 Language Modeling

Language modeling is the task of predicting the next word in a sequence given the previous words. Formally, given a sequence of words $w_1, w_2, ..., w_{t-1}$, the model predicts the probability distribution over the vocabulary for the next word w_t:

$$P(w_t|w_1, w_2, ..., w_{t-1})$$

2.2 Transformer Architecture Fundamentals

2.2.1 Multi-Head Attention Mechanism

The core of the Transformer architecture is the multi-head attention mechanism. Given input sequences, attention computes relationships between all positions simultaneously.

Scaled Dot-Product Attention: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$
where:

- $Q \in \mathbb{R}^{n \times d_k}$ is the query matrix
- $K \in \mathbb{R}^{n \times d_k}$ is the key matrix

- $V \in \mathbb{R}^{n \times d_v}$ is the value matrix
- d_k is the dimension of keys/queries
- n is the sequence length

Multi-Head Attention: $\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$
where each head is computed as: $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$
The projection matrices are:

$$W_i^Q \in \mathbb{R}^{d_{model} \times d_k} \quad (1)$$
$$W_i^K \in \mathbb{R}^{d_{model} \times d_k} \quad (2)$$
$$W_i^V \in \mathbb{R}^{d_{model} \times d_v} \quad (3)$$
$$W^O \in \mathbb{R}^{hd_v \times d_{model}} \quad (4)$$

2.2.2 Position Embeddings

Since attention has no inherent notion of order, position embeddings are added: $\text{PE}(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$ $\text{PE}(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$

2.3 GPT-2 Architecture and Autoregressive Attention

2.3.1 Key Architectural Differences

GPT-2 modifies the standard Transformer architecture for autoregressive generation:

1. Causal (Masked) Self-Attention: Unlike bidirectional attention in BERT, GPT-2 uses causal attention where each position can only attend to previous positions:

$\text{Attention}_{\text{causal}}(Q, K, V) = \text{softmax}\left(\frac{QK^T + M}{\sqrt{d_k}}\right)V$

where M is a lower triangular mask matrix: $M_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$

2. Architectural Components:

- **Decoder-only structure**: Unlike encoder-decoder models
- **Masked multi-head self-attention**: Prevents information leakage
- **Position embeddings**: Learned position embeddings (not sinusoidal)
- **Layer normalization**: Applied before attention and feedforward
- **Residual connections**: Skip connections around each sub-layer

2.3.2 Mathematical Formulation of GPT-2 Block

A single GPT-2 transformer block performs:

Attention Output: $a = \text{LayerNorm}(x) + \text{CausalMultiHead}(\text{LayerNorm}(x))$ Block Output: $y = a + \text{MLP}(\text{LayerNorm}(a))$

where the MLP is: $\text{MLP}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2$

2.3.3 Autoregressive vs Standard Multi-Head Attention

Standard Multi-Head Attention (Bidirectional):

- Each position attends to ALL positions in the sequence
- Attention matrix is full: A_{ij} can be non-zero for all i, j
- Used in BERT-style models for understanding tasks
- Attention weights: $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})}$

Autoregressive (Causal) Attention:

- Each position only attends to previous positions (including itself)
- Attention matrix is lower triangular: $A_{ij} = 0$ if $i < j$
- Used in GPT-style models for generation tasks
- Attention weights: $\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{i} \exp(e_{ik})}$ for $j \leq i$

Implementation Comparison:

```python
# Standard Multi-Head Attention
def standard_attention(Q, K, V):
    scores = torch.matmul(Q, K.transpose(-2, -1)) / sqrt(d_k)
    attn_weights = torch.softmax(scores, dim=-1)
    output = torch.matmul(attn_weights, V)
    return output

# Causal (GPT-2) Attention
def causal_attention(Q, K, V):
    scores = torch.matmul(Q, K.transpose(-2, -1)) / sqrt(d_k)

    # Apply causal mask
    seq_len = scores.size(-1)
    mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))

    scores = scores + mask
    attn_weights = torch.softmax(scores, dim=-1)
    output = torch.matmul(attn_weights, V)
    return output
```

2.3.4 Training Objective

The model is trained to maximize the likelihood of the training data using teacher forcing: $\mathcal{L} = -\sum_{i=1}^{N} \log P(w_i | w_{<i}; \theta)$

where θ represents the model parameters and $w_{<i}$ denotes all tokens before position i.

2.3.5 Attention Pattern Analysis

The attention patterns differ significantly:

Bidirectional Attention Matrix: $A_{\text{bidirectional}} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix}$

Causal Attention Matrix: $A_{\text{causal}} = \begin{pmatrix} \alpha_{11} & 0 & 0 & \cdots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \cdots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix}$

2.4 Fine-tuning Process

Fine-tuning involves taking a pre-trained model and adapting it to a specific task or domain by continuing training on task-specific data with a smaller learning rate.

3 Mathematical Implementation Details

3.1 Complete GPT-2 Forward Pass

The complete mathematical formulation of a GPT-2 forward pass:

Input Processing:

$$\text{Input tokens: } \mathbf{x} = [x_1, x_2, ..., x_n] \tag{5}$$
$$\text{Token embeddings: } \mathbf{E}_{\text{tok}} = \text{Embedding}(\mathbf{x}) \tag{6}$$
$$\text{Position embeddings: } \mathbf{E}_{\text{pos}} = \text{PositionEmbedding}([1, 2, ..., n]) \tag{7}$$
$$\text{Initial hidden state: } \mathbf{h}_0 = \mathbf{E}_{\text{tok}} + \mathbf{E}_{\text{pos}} \tag{8}$$

Transformer Block Operations: For each layer $l = 1, 2, ..., L$:

$$\text{Pre-attention norm: } \mathbf{h}_l^{(1)} = \text{LayerNorm}(\mathbf{h}_{l-1}) \tag{9}$$
$$\text{Multi-head attention: } \mathbf{a}_l = \text{CausalMultiHead}(\mathbf{h}_l^{(1)}) \tag{10}$$
$$\text{Post-attention residual: } \mathbf{h}_l^{(2)} = \mathbf{h}_{l-1} + \mathbf{a}_l \tag{11}$$
$$\text{Pre-MLP norm: } \mathbf{h}_l^{(3)} = \text{LayerNorm}(\mathbf{h}_l^{(2)}) \tag{12}$$
$$\text{MLP computation: } \mathbf{m}_l = \text{MLP}(\mathbf{h}_l^{(3)}) \tag{13}$$
$$\text{Post-MLP residual: } \mathbf{h}_l = \mathbf{h}_l^{(2)} + \mathbf{m}_l \tag{14}$$

Output Generation:

$$\text{Final normalization: } \mathbf{h}_{\text{final}} = \text{LayerNorm}(\mathbf{h}_L) \tag{15}$$
$$\text{Logits: } \textbf{logits} = \mathbf{h}_{\text{final}} \mathbf{W}_{\text{lm\_head}} \tag{16}$$
$$\text{Probabilities: } \mathbf{p} = \text{softmax}(\textbf{logits}) \tag{17}$$

3.2 Attention Mechanism Deep Dive

3.2.1 Query, Key, Value Computation

For each attention head i in layer l:

$$\mathbf{Q}_i = \mathbf{h}_l^{(1)}\mathbf{W}_i^Q + \mathbf{b}_i^Q \tag{18}$$
$$\mathbf{K}_i = \mathbf{h}_l^{(1)}\mathbf{W}_i^K + \mathbf{b}_i^K \tag{19}$$
$$\mathbf{V}_i = \mathbf{h}_l^{(1)}\mathbf{W}_i^V + \mathbf{b}_i^V \tag{20}$$

where $\mathbf{W}_i^Q, \mathbf{W}_i^K, \mathbf{W}_i^V \in \mathbb{R}^{d_{model} \times d_k}$ and $d_k = d_{model}/h$.

3.2.2 Causal Attention Computation

The causal attention for head i is computed as:

$$\mathbf{S}_i = \frac{\mathbf{Q}_i\mathbf{K}_i^T}{\sqrt{d_k}} \tag{21}$$
$$\mathbf{M}_{jk} = \begin{cases} 0 & \text{if } j \geq k \\ -\infty & \text{if } j < k \end{cases} \tag{22}$$
$$\mathbf{A}_i = \text{softmax}(\mathbf{S}_i + \mathbf{M}) \tag{23}$$
$$\mathbf{O}_i = \mathbf{A}_i\mathbf{V}_i \tag{24}$$

3.2.3 Multi-Head Combination

The outputs from all heads are concatenated and projected:

$$\mathbf{O}_{\text{concat}} = \text{Concat}(\mathbf{O}_1, \mathbf{O}_2, ..., \mathbf{O}_h) \tag{25}$$
$$\mathbf{O}_{\text{final}} = \mathbf{O}_{\text{concat}}\mathbf{W}^O + \mathbf{b}^O \tag{26}$$

3.3 MLP Layer Mathematics

The MLP in each transformer block:

$$\mathbf{h}_{\text{intermediate}} = \text{GELU}(\mathbf{h}_l^{(3)}\mathbf{W}_1 + \mathbf{b}_1) \tag{27}$$
$$\mathbf{h}_{\text{output}} = \mathbf{h}_{\text{intermediate}}\mathbf{W}_2 + \mathbf{b}_2 \tag{28}$$

where $\mathbf{W}_1 \in \mathbb{R}^{d_{model} \times 4d_{model}}$ and $\mathbf{W}_2 \in \mathbb{R}^{4d_{model} \times d_{model}}$.
GELU Activation: $\text{GELU}(x) = x \cdot \Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$

3.4 Layer Normalization

GPT-2 uses layer normalization before each sub-layer:

$$\mu = \frac{1}{d_{model}} \sum_{i=1}^{d_{model}} x_i \tag{29}$$

$$\sigma^2 = \frac{1}{d_{model}} \sum_{i=1}^{d_{model}} (x_i - \mu)^2 \tag{30}$$

$$\text{LayerNorm}(x) = \gamma \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{31}$$

where γ and β are learnable parameters.

3.5 Training Loss and Optimization

3.5.1 Cross-Entropy Loss

For next token prediction, the loss at each position is:
$\mathcal{L}_i = -\log P(w_i|w_{<i}) = -\log \frac{\exp(\textbf{logits}_{i,w_i})}{\sum_{v \in V} \exp(\textbf{logits}_{i,v})}$
The total loss is: $\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i$

3.5.2 Gradient Computation

The gradient of the loss with respect to logits:
$\frac{\partial \mathcal{L}}{\partial \textbf{logits}_{i,v}} = P(v|w_{<i}) - \mathbf{1}_{v=w_i}$
where $\mathbf{1}_{v=w_i}$ is 1 if v is the true token and 0 otherwise.

3.6 Comparison: Standard vs Causal Attention

Aspect	**Standard (BERT)**	**Causal (GPT-2)**
Attention Matrix	Full matrix	Lower triangular
Information Flow	Bidirectional	Unidirectional
Masking	Random tokens	Future tokens
Training Objective	MLM + NSP	Next token prediction
Use Case	Understanding	Generation
Computational Cost	$O(n^2)$	$O(n^2)$
Memory Pattern	Full attention	Causal attention

Table 1: Comparison of Attention Mechanisms

3.7 Implementation in PyTorch

Here's how the core attention mechanisms are implemented:

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class CausalMultiHeadAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

        # Create causal mask
        self.register_buffer('mask', torch.triu(torch.ones(512, 512), diagonal=1))

    def forward(self, x):
        batch_size, seq_len, d_model = x.size()

        # Compute Q, K, V
        Q = self.W_q(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        K = self.W_k(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)
        V = self.W_v(x).view(batch_size, seq_len, self.n_heads, self.d_k).transpose(1, 2)

        # Compute attention scores
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)

        # Apply causal mask
        mask = self.mask[:seq_len, :seq_len]
        scores = scores.masked_fill(mask == 1, float('-inf'))

        # Apply softmax
        attn_weights = F.softmax(scores, dim=-1)

        # Apply attention to values
        out = torch.matmul(attn_weights, V)

        # Concatenate heads
        out = out.transpose(1, 2).contiguous().view(batch_size, seq_len, d_model)

        # Final linear transformation
        out = self.W_o(out)

        return out

class GPT2Block(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
```

```python
        self.ln_1 = nn.LayerNorm(d_model)
        self.attn = CausalMultiHeadAttention(d_model, n_heads)
        self.ln_2 = nn.LayerNorm(d_model)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model)
        )

    def forward(self, x):
        # Pre-norm architecture
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlp(self.ln_2(x))
        return x
```

3.8 Import Statements

The code begins with comprehensive imports:

```python
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from transformers import (
    GPT2LMHeadModel, GPT2Tokenizer,
    TrainingArguments, Trainer,
    DataCollatorForLanguageModeling
)
from datasets import load_dataset
# ... additional imports
```

These imports provide:

- **torch**: Core PyTorch functionality
- **transformers**: Hugging Face's transformer models
- **datasets**: Dataset loading utilities
- **Standard libraries**: For data processing and visualization

4 Dataset Implementation

4.1 WikiTextDataset Class

The `WikiTextDataset` class handles data preprocessing for the WikiText-2 dataset:

```python
class WikiTextDataset(Dataset):
    def __init__(self, texts: List[str], tokenizer, max_length: int =
    512):
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.examples = []

        for text in tqdm(texts):
            if len(text.strip()) > 0:
                tokens = tokenizer.encode(text, add_special_tokens=True
    )
```

```python

                # Create overlapping sequences
                for i in range(0, len(tokens) - 1, max_length // 2):
                    chunk = tokens[i:i + max_length]
                    if len(chunk) > 1:
                        self.examples.append(chunk)
```

4.1.1 Key Features

1. **Tokenization**: Converts text to token IDs using GPT-2 tokenizer
2. **Sliding Window**: Creates overlapping sequences with 50% overlap
3. **Length Filtering**: Ensures sequences have minimum length for training

4.1.2 Data Flow

Algorithm 1 Dataset Creation Process

```
for each text in input texts do
  if text is not empty then
    tokenize text to get token IDs
    for i = 0 to length(tokens) - 1 step max_length/2 do
      chunk = tokens[i:i + max_length]
      if length(chunk) ¿ 1 then
        add chunk to examples
      end if
    end for
  end if
end for
```

4.2 Data Loading Functions

The `load_and_prepare_data()` function handles dataset loading:

```python
def load_and_prepare_data():
    dataset = load_dataset('wikitext', 'wikitext-2-raw-v1')

    train_texts = [text for text in dataset['train']['text']
                   if len(text.strip()) > 0]
    valid_texts = [text for text in dataset['validation']['text']
                   if len(text.strip()) > 0]
    test_texts = [text for text in dataset['test']['text']
                  if len(text.strip()) > 0]

    return train_texts, valid_texts, test_texts
```

This function:

- Loads WikiText-2 dataset from Hugging Face
- Filters empty texts
- Returns train/validation/test splits

5 Model Setup and Configuration

5.1 Model and Tokenizer Initialization

The `setup_model_and_tokenizer()` function configures the GPT-2 components:

```python
def setup_model_and_tokenizer():
    model_name = "gpt2"

    tokenizer = GPT2Tokenizer.from_pretrained(model_name)
    model = GPT2LMHeadModel.from_pretrained(model_name)

    # Add padding token
    tokenizer.pad_token = tokenizer.eos_token

    return model, tokenizer
```

5.1.1 Key Configurations

- **Model**: GPT2LMHeadModel with language modeling head
- **Tokenizer**: GPT2Tokenizer for text-to-token conversion
- **Padding**: Uses EOS token as padding token

6 Training Pipeline

6.1 Fine-tuning Function

The `fine_tune_model()` function implements the training loop:

```python
def fine_tune_model(model, tokenizer, train_dataset, eval_dataset,
                    output_dir="./gpt2-wikitext2"):

    data_collator = DataCollatorForLanguageModeling(
        tokenizer=tokenizer,
        mlm=False,  # GPT-2 is autoregressive
        pad_to_multiple_of=8,
        return_tensors="pt"
    )

    training_args = TrainingArguments(
        output_dir=output_dir,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        per_device_eval_batch_size=4,
        warmup_steps=500,
        learning_rate=5e-5,
        weight_decay=0.01,
        fp16=True,
        # ... additional arguments
    )

    trainer = Trainer(
        model=model,
```

```
        args=training_args,
        data_collator=data_collator,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer,
    )

    trainer.train()
    return trainer
```

6.2 Training Arguments Analysis

Parameter	Description
num_train_epochs	Number of training epochs (3)
per_device_train_batch_size	Training batch size per device (4)
warmup_steps	Learning rate warmup steps (500)
learning_rate	Initial learning rate (5e-5)
weight_decay	L2 regularization coefficient (0.01)
fp16	Mixed precision training (True)
gradient_accumulation_steps	Gradient accumulation (2)

Table 2: Key Training Parameters

6.3 Data Collator

The `DataCollatorForLanguageModeling` handles:

- Padding sequences to equal length
- Creating attention masks
- Preparing labels for next token prediction

7 Evaluation Framework

7.1 NWPEvaluator Class

The `NWPEvaluator` class provides comprehensive evaluation metrics:

```
class NWPEvaluator:
    def __init__(self, model, tokenizer, device):
        self.model = model
        self.tokenizer = tokenizer
        self.device = device
        self.model.eval()

    def calculate_perplexity(self, dataloader) -> float:
        # Implementation for perplexity calculation

    def calculate_top_k_accuracy(self, dataloader,
```

```
                                 k_values: List[int] = [1, 5, 10]) ->
    Dict[int, float]:
        # Implementation for top-k accuracy

    def sample_predictions(self, text: str,
                           num_predictions: int = 5) -> List[str]:
        # Implementation for sample predictions
```

7.2 Perplexity Calculation

Perplexity measures how well the model predicts the next word:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_{<i})\right)$$

Implementation:

```
def calculate_perplexity(self, dataloader) -> float:
    total_loss = 0
    total_tokens = 0

    with torch.no_grad():
        for batch in dataloader:
            input_ids = batch['input_ids'].to(self.device)
            attention_mask = batch['attention_mask'].to(self.device)

            # Shift labels for next token prediction
            labels = input_ids.clone()
            labels[:, :-1] = input_ids[:, 1:]
            labels[:, -1] = -100  # Ignore last token

            outputs = self.model(input_ids=input_ids,
                                 attention_mask=attention_mask,
                                 labels=labels)

            loss = outputs.loss
            valid_tokens = (labels != -100).sum().item()

            total_loss += loss.item() * valid_tokens
            total_tokens += valid_tokens

    avg_loss = total_loss / total_tokens
    perplexity = math.exp(avg_loss)
    return perplexity
```

7.3 Top-k Accuracy

Top-k accuracy measures if the correct next word appears in the top-k predictions:

```
def calculate_top_k_accuracy(self, dataloader, k_values: List[int] =
    [1, 5, 10]):
    correct_predictions = {k: 0 for k in k_values}
    total_predictions = 0

    with torch.no_grad():
        for batch in dataloader:
```

```
            # Process each sequence
            for seq_idx in range(input_ids.size(0)):
                sequence = input_ids[seq_idx]

                # Predict each position
                for pos in range(1, valid_length):
                    context = sequence[:pos].unsqueeze(0)
                    target = sequence[pos].item()

                    outputs = self.model(context)
                    logits = outputs.logits[0, -1, :]

                    # Get top-k predictions
                    top_k_tokens = torch.topk(logits, max(k_values)).
    indices

                    # Check accuracy for different k values
                    for k in k_values:
                        if target in top_k_tokens[:k]:
                            correct_predictions[k] += 1

                    total_predictions += 1

        accuracies = {k: correct_predictions[k] / total_predictions for k
    in k_values}
        return accuracies
```

8 Data Exploration Tools

8.1 Training Data Inspection

The code includes comprehensive data exploration functions:

8.1.1 inspect_training_examples()

This function provides detailed analysis of training examples:

- Sequence length statistics
- Token-level breakdown
- Text representation with special tokens
- Attention mask visualization

8.1.2 show_next_word_prediction_examples()

Demonstrates the input-output format for next word prediction:

```
def show_next_word_prediction_examples(dataset, tokenizer, num_examples
    =3):
    for i in range(min(num_examples, len(dataset))):
        example = dataset[i]
        input_ids = example['input_ids']

```

```
        # Show first 10 positions as input -> target pairs
        for pos in range(1, min(11, len(input_ids))):
            context_ids = input_ids[:pos]
            target_id = input_ids[pos]

            context_text = tokenizer.decode(context_ids,
    skip_special_tokens=True)
            target_text = tokenizer.decode([target_id],
    skip_special_tokens=True)

            print(f"'{context_text}' -> '{target_text}'")
```

8.2 Statistical Analysis

The `analyze_dataset_statistics()` function provides:

- Sequence length distribution
- Token frequency analysis
- Vocabulary statistics
- Data quality metrics

9 Main Execution Pipeline

9.1 Main Function

The `main()` function orchestrates the entire pipeline:

Algorithm 2 Main Execution Flow

Set device (GPU/CPU)
Load and prepare data
Setup model and tokenizer
Create datasets
Fine-tune model
Evaluate model
Generate sample predictions
Create results summary

9.2 Results Summary

The pipeline generates comprehensive results including:

- Perplexity scores
- Top-k accuracy metrics
- Sample predictions
- Performance visualizations

10 Advanced Topics

10.1 Attention Visualization

Understanding attention patterns is crucial for interpreting model behavior:

```
def visualize_attention_patterns(model, tokenizer, text):
    """Visualize attention patterns for a given text"""
    inputs = tokenizer(text, return_tensors="pt")

    with torch.no_grad():
        outputs = model(**inputs, output_attentions=True)
        attentions = outputs.attentions  # List of attention tensors

    # attentions[i] has shape: [batch_size, num_heads, seq_len, seq_len
    ]
    # For layer i, head j: attentions[i][0, j, :, :]

    import matplotlib.pyplot as plt

    # Visualize attention for first layer, first head
    attn_matrix = attentions[0][0, 0, :, :].numpy()

    plt.figure(figsize=(10, 8))
    plt.imshow(attn_matrix, cmap='Blues')
    plt.colorbar()
    plt.title("Causal Attention Pattern (Layer 0, Head 0)")
    plt.xlabel("Key Position")
    plt.ylabel("Query Position")
    plt.show()
```

10.2 Temperature and Sampling Strategies

The code can be extended with different sampling strategies:

```
def sample_with_temperature(logits, temperature=1.0):
    """Sample from logits with temperature scaling"""
    if temperature == 0:
        return torch.argmax(logits, dim=-1)

    # Apply temperature scaling
    scaled_logits = logits / temperature
    probs = F.softmax(scaled_logits, dim=-1)

    # Sample from the distribution
    return torch.multinomial(probs, 1)

def top_k_top_p_sampling(logits, top_k=50, top_p=0.9, temperature=1.0):
    """Combined top-k and top-p (nucleus) sampling"""
    # Apply temperature
    logits = logits / temperature

    # Top-k filtering
    if top_k > 0:
        top_k_logits, top_k_indices = torch.topk(logits, top_k)
        logits_filtered = torch.full_like(logits, float('-inf'))
        logits_filtered.scatter_(1, top_k_indices, top_k_logits)
```

```
            logits = logits_filtered

    # Top-p filtering
    if top_p < 1.0:
        sorted_logits, sorted_indices = torch.sort(logits, descending=True)
        cumulative_probs = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)

        # Remove tokens with cumulative probability above threshold
        sorted_indices_to_remove = cumulative_probs > top_p
        sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()
        sorted_indices_to_remove[..., 0] = 0

        indices_to_remove = sorted_indices_to_remove.scatter(1, sorted_indices, sorted_indices_to_remove)
        logits = logits.masked_fill(indices_to_remove, float('-inf'))

    # Sample from filtered distribution
    probs = F.softmax(logits, dim=-1)
    return torch.multinomial(probs, 1)
```

10.3 Memory-Efficient Training

For large-scale training, memory optimization is crucial:

```
from torch.utils.checkpoint import checkpoint

class MemoryEfficientGPT2Block(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.ln_1 = nn.LayerNorm(d_model)
        self.attn = CausalMultiHeadAttention(d_model, n_heads)
        self.ln_2 = nn.LayerNorm(d_model)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Linear(d_ff, d_model)
        )

    def forward(self, x):
        # Use gradient checkpointing to save memory
        def attention_forward(x):
            return self.attn(self.ln_1(x))

        def mlp_forward(x):
            return self.mlp(self.ln_2(x))

        # Apply gradient checkpointing
        x = x + checkpoint(attention_forward, x)
        x = x + checkpoint(mlp_forward, x)
        return x
```

10.4 Distributed Training Considerations

For multi-GPU training, the code can be extended:

```python
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup_distributed_training():
    """Setup for distributed training across multiple GPUs"""
    if torch.cuda.is_available() and torch.cuda.device_count() > 1:
        dist.init_process_group(backend='nccl')
        local_rank = int(os.environ['LOCAL_RANK'])
        torch.cuda.set_device(local_rank)

        # Wrap model with DDP
        model = DDP(model, device_ids=[local_rank])

        # Use DistributedSampler for data loading
        sampler = torch.utils.data.distributed.DistributedSampler(
    dataset)
        dataloader = DataLoader(dataset, sampler=sampler, batch_size=
    batch_size)

        return model, dataloader

    return model, dataloader
```

10.5 Basic Usage

To run the complete pipeline:

```python
# Run full pipeline
model, tokenizer, results = main()

# Explore data first
sample_train, sample_valid, tokenizer = explore_training_data()
```

10.6 Custom Configuration

For custom training configurations:

```python
# Custom dataset size
train_dataset = WikiTextDataset(train_texts[:5000], tokenizer)
eval_dataset = WikiTextDataset(valid_texts[:1000], tokenizer)

# Custom training arguments
training_args = TrainingArguments(
    output_dir="./custom-gpt2",
    num_train_epochs=5,
    per_device_train_batch_size=8,
    learning_rate=3e-5,
    # ... other parameters
)
```

11 Performance Considerations

11.1 Memory Management

The code includes several memory optimization strategies:

- Mixed precision training (fp16)
- Gradient accumulation
- Efficient data loading
- Batch size optimization

11.2 Computational Requirements

Component	**Requirement**
GPU Memory	8GB+ recommended
Training Time	2-4 hours (depends on data size)
Disk Space	2GB+ for model and data
RAM	16GB+ recommended

Table 3: Computational Requirements

12 Best Practices and Recommendations

12.1 Training Tips

1. Start with small dataset subsets for testing
2. Monitor training loss and validation metrics
3. Use appropriate learning rates (typically 1e-5 to 5e-5)
4. Implement early stopping for overfitting prevention

12.2 Evaluation Guidelines

1. Use multiple evaluation metrics
2. Test on held-out data
3. Compare with baseline models
4. Analyze failure cases

13 Conclusion

This GPT-2 fine-tuning pipeline provides a comprehensive framework for next word prediction tasks. The code implements best practices for:

- Data preprocessing and tokenization
- Model training and optimization

- Comprehensive evaluation
- Result analysis and visualization

The modular design allows for easy customization and extension for specific use cases while maintaining robust performance and reliability.

13.1 Future Enhancements

Potential improvements include:

- Distributed training support
- Advanced regularization techniques
- Real-time inference optimization
- Integration with larger language models

14 References

1. Radford, A., et al. (2019). Language models are unsupervised multitask learners.
2. Vaswani, A., et al. (2017). Attention is all you need. NIPS.
3. Wolf, T., et al. (2020). Transformers: State-of-the-art natural language processing.
4. Merity, S., et al. (2016). Pointer sentinel mixture models.