Kathmandu University

Department of Computer Science and Engineering

Dhulikhel,Kavre

Mini Project:

Transformer Based Next Word Prediction with Explainable AI (xAI)

COMP: 472

CE-IV/I

Submitted by:

Aakriti Poudel(43)

Siddharth Chaudhary(11)

Submitted to:

Sanjog Sigdel

Department of Computer Science and Engineering

Date of Submission:

2025/06/30

1. Introduction

Language modeling, the task of predicting the next word in a sequence, has been a cornerstone of Natural Language Processing (NLP) for decades. Early advancements in this field were largely driven by Recurrent Neural Networks (RNNs) and their more sophisticated variants, Long Short-Term Memory (LSTM) networks. These architectures excel at processing sequential data by maintaining an internal state that captures information from previous tokens, finding applications in diverse areas such as machine translation, speech recognition, and text generation.

A pivotal moment in the evolution of language modeling arrived with the publication of the paper "Attention Is All You Need" by Vaswani et al. (2017). This groundbreaking work introduced the Transformer architecture, which revolutionized the domain by entirely foregoing recurrence and convolutions in favor of a novel mechanism: self-attention. The self-attention mechanism allowed the model to weigh the importance of every other token in the input sequence when processing a specific token, enabling the capture of long-range dependencies more effectively. Crucially, its inherent parallelism significantly accelerated training times for large models, paving the way for the development of highly powerful and efficient language models.

This project aims to delve into the core concepts of Transformer architectures by building a next-word predictor from scratch. Furthermore, a key focus is to integrate Explainable AI (XAI) techniques directly into the model, providing transparency into its internal decision-making processes, which is often a challenge with complex neural networks.

2. Objective

The main objectives of our project is given below:

- To custom-build a Transformer-based next-word predictor from scratch, understanding its foundational components and operational flow.
- To integrate and apply Explainable AI concepts using perturbation-based feature attribution and approximate SHAP (Linear SHAP) to elucidate the model's predictions.

3. Background

3.1 A Brief History of Language Models

The journey of language modeling has seen significant milestones. Early statistical models, like N-grams, estimated word probabilities based on preceding words. The advent of neural networks brought about Recurrent Neural Networks (RNNs), which processed sequences one element at a time, maintaining a hidden state that captured context. While powerful, RNNs struggled with long-range dependencies due to vanishing or exploding gradients. Long Short-Term Memory (LSTM) networks were introduced to mitigate these issues with their gating mechanisms, allowing them to selectively remember or forget information over long sequences. LSTMs became the state-of-the-art for sequence modeling tasks for a considerable period.

3.2 Transformer Theory

The Transformer architecture, unlike RNNs and LSTMs, processes entire sequences in parallel, relying solely on attention mechanisms.

3.2.1 Self-Attention

This is the core building block. For each token in an input sequence, self-attention allows the model to compute a weighted sum of all other tokens, where the weights are dynamically calculated based on their relevance to the current token. This process involves creating three learned linear transformations for each input token: a Query (Q), a Key (K), and a Value (V). The attention scores are computed by taking the dot product of the Query with all Keys, scaling by √dk (the square root of the dimension of the keys for stability), and applying a softmax function to get the attention weights. These weights are then used to sum the Values.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

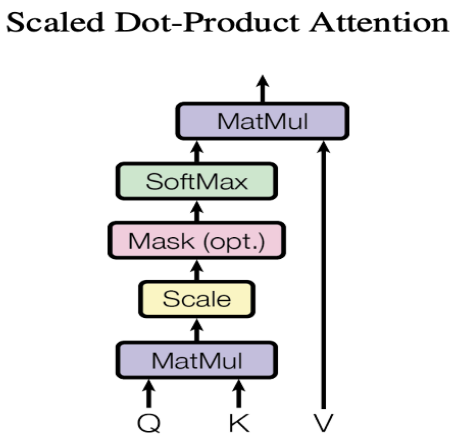


Fig: Scaled Dot-Product Attention

3.2.2 Multi-Head Attention

Instead of performing a single attention function, Multi-Head Attention performs this process multiple times in parallel with different linear projections for Q,K,V. The results from each "head" are then concatenated and linearly transformed. This allows the model to jointly attend to information from different representation subspaces at different positions, enriching its understanding of relationships within the sequence.

$$\begin{aligned}\mathrm{MultiHead}(Q, K, V) &= \mathrm{Concat}(\mathrm{head_1}, ..., \mathrm{head_h})W^O \\ \text{where}\ \mathrm{head_i} &= \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

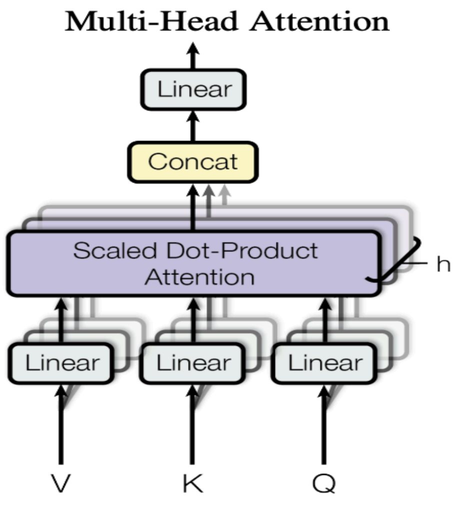


Fig: Multi-Head Attention consists of several attention layers running in parallel

3.2.2 Feed-Forward Network (FFN)

After the attention mechanism, the output of each position passes through an identical, independently applied position-wise feed-forward network. This typically consists of two linear transformations with a ReLU activation in between.

3.2.3 Block

A single Transformer "block" consists of a Multi-Head Attention layer followed by a Feed-Forward Network. Both sub-layers are wrapped with residual connections and LayerNormalization. Transformers are typically built by stacking multiple such blocks, allowing the model to learn increasingly complex representations.

3.2.4 Generation (Inference)

During inference for next-word prediction, the Transformer model takes an input sequence and predicts the probability distribution over the vocabulary for the next token. This process can be auto-regressive, where the predicted token is appended to the input sequence for subsequent predictions.

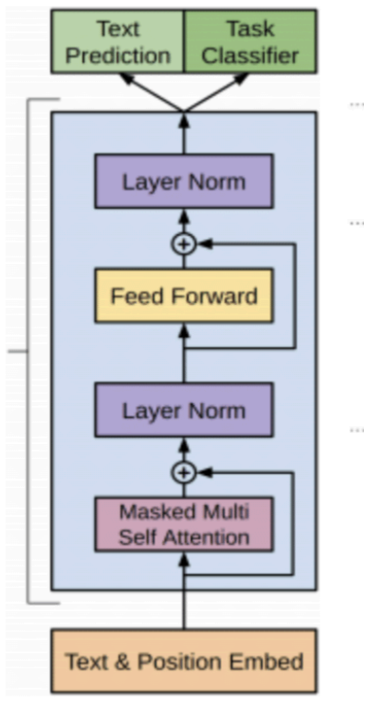


Fig: Decoder-Only Transformer Architecture

3.3 Explainable AI Methods

Our project utilizes two key XAI methods to understand the Transformer's predictions:

3.3.1 Perturbation-Based Feature Attribution

This method operates on the fundamental principle of observing how the model's output changes when specific parts of its input are altered or "perturbed." The core idea is: "If I remove or change this token, how much does the prediction change?"

Step-by-Step Process:

1. Baseline Prediction: Obtain the model's original prediction and its confidence score for the next token given the full input sequence x.

2. Perturbation: For each token at position i in the input sequence, create a modified version x′ where token i is either:
   - Masked: Replaced with a special [MASK] token (or a 0 token ID in our implementation), effectively removing its information.
   - Randomized: Replaced with a randomly sampled token from the vocabulary, introducing noise.
3. Impact Measurement: Run the model on the perturbed sequence x′ and measure the new confidence for the original predicted next token. The "impact" is the difference between the baseline confidence and the perturbed confidence.

4. Averaging: Repeat the perturbation and impact measurement multiple times (n_perturbations) for each token. The final importance score for token i (ϕi) is the average of these impacts:
   ϕi=Baseline Confidence−E[Confidence after perturbing token i].
   This score quantifies how much the presence of a token contributes to the model's confidence in its prediction. Positive scores indicate helpful tokens, while negative scores indicate harmful ones.

3.3.2 SHAP (SHapley Additive exPlanations)

SHAP values are a theoretically sound method derived from cooperative game theory, aiming to fairly distribute the "credit" for a model's prediction among its input features. The fundamental question SHAP answers is: "What is the fair contribution of this token across all possible contexts?"

Perturbation is actually a special case of SHAP which only considers a single coalition (i.e token i joining all other tokens). The exact calculation of Shapley values involves computing the marginal contribution of a feature across all possible subsets (coalitions) of other features. This makes it computationally intensive, especially for sequences of length n (requiring 2n calculations).

Approximate SHAP (Linear SHAP)

To make SHAP practical for language models, an approximation is often used. In this project, we have used Linear SHAP. This method estimates Shapley values by:

1. **Sampling Coalitions:** Instead of evaluating all possible 2n subsets, a fixed number of random subsets of tokens (coalitions) are sampled.

2. **Perturbation:** For each sampled coalition, a perturbed sequence x~(j) is created. Tokens *present* in the coalition are kept as they are, while tokens *not present* are perturbed (masked or randomized).

   $$x\sim(j)=x\odot z(j)+p\odot(1-z(j))$$

   where:

   z(j) is a binary mask representing the coalition, and p is the perturbation baseline.

3. **Model Output Collection:** The model's prediction confidence for the target output is measured for each perturbed sequence x~(j), resulting in an output change c(j)=y^(j)−y^baseline.

4. **Linear Regression:** A simple linear regression model is then fit to the collected data, where the input is the coalition mask z(j) and the output is the confidence change c(j).

The coefficients of this linear model become the approximate Shapley values (ϕ), representing the estimated contribution of each token. $c(j) \approx z(j) \cdot \phi$

This approximation allows for a much faster computation of contribution scores that are still grounded in the fairness principles of Shapley values.

4. Literature Review

This project builds upon foundational advancements in deep learning and interpretability. The Transformer architecture, introduced by Vaswani et al. (2017) in "Attention Is All You Need," revolutionized sequence modeling by exclusively relying on self-attention mechanisms, enabling unprecedented parallelization and leading to powerful models like GPT (Radford et al., 2018) and BERT (Devlin et al., 2019). While the trend has often favored Large Language Models (LLMs), this work focuses on Small Language Models (SLMs), which offer computational efficiency and practical advantages. Research on SLMs, as highlighted by scaling laws (Kaplan et al., 2020) and recent surveys (Subramanian et al., 2025; Zhao et al., 2025), demonstrates their capability for various tasks, aligning with the custom Transformer implementation in this project.

To address the inherent "black-box" nature of such models, Explainable Artificial Intelligence (XAI) techniques are crucial. This project employs perturbation-based feature attribution methods, including SHAP (Lundberg & Lee, 2017), which provides a unified, game-theoretic approach to attribute prediction contributions to individual features. Other influential methods like LIME (Ribeiro et al., 2016) and Integrated Gradients (Sundararajan et al., 2017) also contribute to the field of XAI by offering local explanations through input perturbations or gradient-based attributions, as broadly surveyed in works on perturbation-based methods (Doshi-Velez & Kim, 2017). By integrating these XAI approaches with a custom SLM, this project aims to enhance transparency and understanding of model behavior.

5. Methodology

5.1 Data Collection

Our dataset was meticulously compiled from a diverse range of sources to provide rich and varied linguistic patterns:

- arXiv Research Papers: Scientific texts, offering formal and technical language licensed as CC by 4.0.
- Classic Books with No Copyright (Project Gutenberg): A wide array of literary styles, from historical fiction to philosophical works.
- Our AI Textbook (AI: A Modern Approach by Stuart Russell and Peter Norvig): Specialized technical language pertaining to Artificial Intelligence.
- A Fiction Book (Singularity Is Near by Ray Kurzweil): Contemporary non-fiction, often containing complex sentence structures and modern terminology.

5.2 Data Tokenization

The collected raw text data undergoes tokenization using Google SentencePiece. SentencePiece is a subword tokenizer that trains directly on raw text, enabling it to handle out-of-vocabulary words by breaking them down into known subword units. This approach is crucial for robust language modeling, especially with diverse text sources.

5.3 Data Preparation

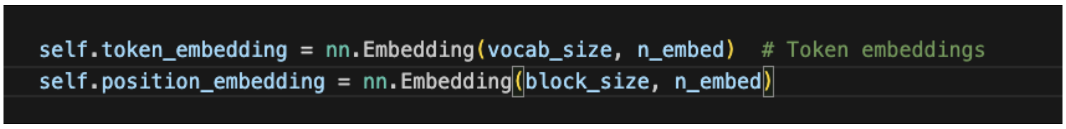
Once tokenized, the numerical token IDs are processed further:

- The entire tokenized dataset is split into training and testing sets to ensure proper model evaluation on unseen data.
- Using PyTorch's Dataset and DataLoader utilities, the data is organized into batches. Importantly, these batches are created with certain block_size (sequence length)

5.4 Transformer Architecture Implementation

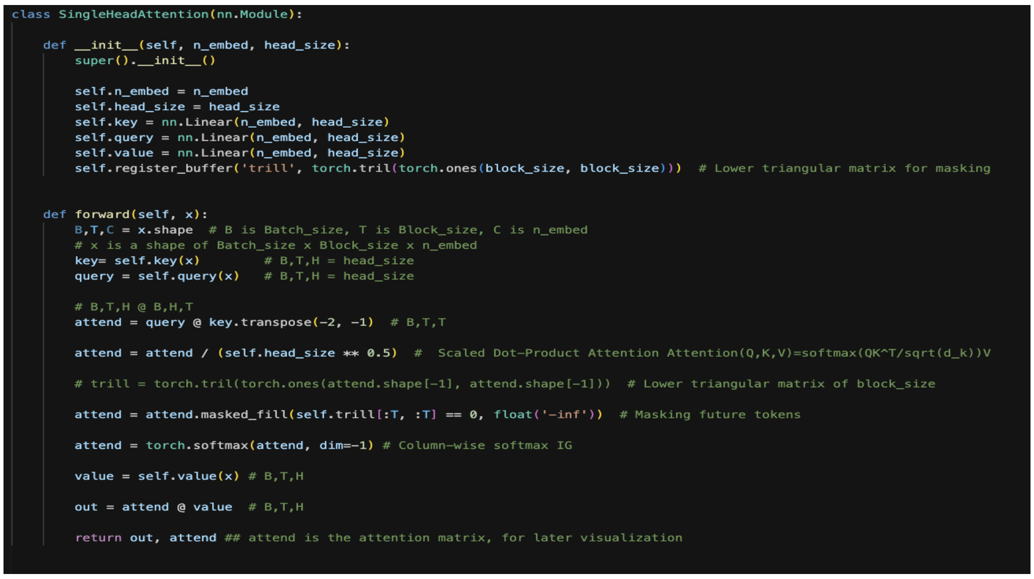
Our Transformer model is custom-built, adhering to the principles laid out in "Attention Is All You Need":

- **Dimensional (Input) Embedding and Positional Encoding:**
  Each input token is first converted into a fixed-size dense vector representation. This is done through an embedding layer, which maps discrete token IDs to continuous vectors.Since the self-attention mechanism is permutation-invariant (meaning it doesn't inherently understand the order of tokens), positional encodings are added to the input embeddings.



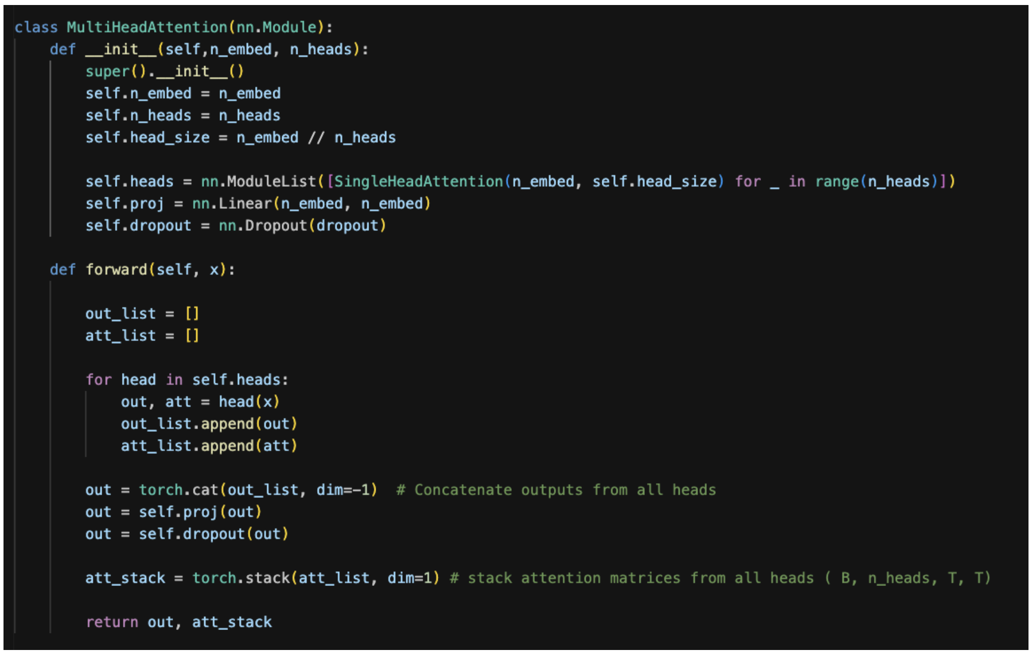
```
self.token_embedding = nn.Embedding(vocab_size, n_embed)  # Token embeddings
self.position_embedding = nn.Embedding(block_size, n_embed)
```

- **Self-Attention:**
  Implemented as the fundamental mechanism to compute attention scores within a single sequence.



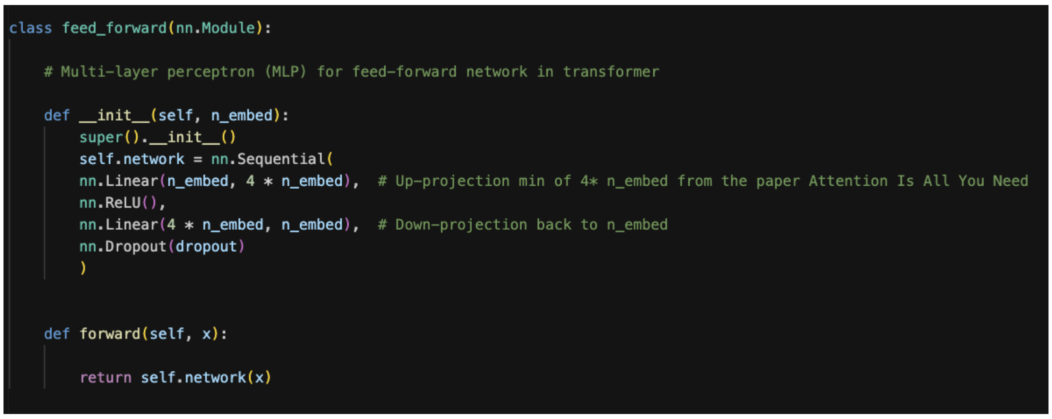
```
class SingleHeadAttention(nn.Module):

    def __init__(self, n_embed, head_size):
        super().__init__()

        self.n_embed = n_embed
        self.head_size = head_size
        self.key = nn.Linear(n_embed, head_size)
        self.query = nn.Linear(n_embed, head_size)
        self.value = nn.Linear(n_embed, head_size)
        self.register_buffer('trill', torch.tril(torch.ones(block_size, block_size)))  # Lower triangular matrix for masking


    def forward(self, x):
        B,T,C = x.shape  # B is Batch_size, T is Block_size, C is n_embed
        # x is a shape of Batch_size x Block_size x n_embed
        key= self.key(x)        # B,T,H = head_size
        query = self.query(x)   # B,T,H = head_size

        # B,T,H @ B,H,T
        attend = query @ key.transpose(-2, -1)  # B,T,T

        attend = attend / (self.head_size ** 0.5)  #  Scaled Dot-Product Attention Attention(Q,K,V)=softmax(QK^T/sqrt(d_k))V

        # trill = torch.tril(torch.ones(attend.shape[-1], attend.shape[-1]))  # Lower triangular matrix of block_size

        attend = attend.masked_fill(self.trill[:T, :T] == 0, float('-inf'))  # Masking future tokens

        attend = torch.softmax(attend, dim=-1) # Column-wise softmax IG

        value = self.value(x) # B,T,H

        out = attend @ value  # B,T,H

        return out, attend ## attend is the attention matrix, for later visualization
```

- **Multi-Head Attention:** Multiple self-attention mechanisms running in parallel, with their outputs concatenated and projected, allowing the model to focus on different aspects of the input.



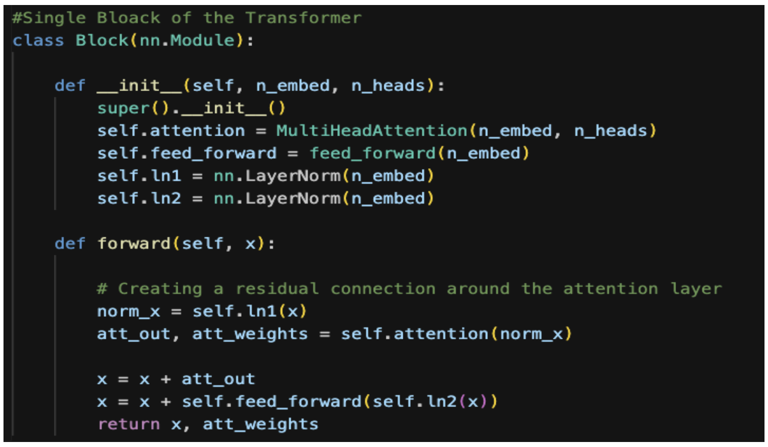
```python
class MultiHeadAttention(nn.Module):
    def __init__(self,n_embed, n_heads):
        super().__init__()
        self.n_embed = n_embed
        self.n_heads = n_heads
        self.head_size = n_embed // n_heads

        self.heads = nn.ModuleList([SingleHeadAttention(n_embed, self.head_size) for _ in range(n_heads)])
        self.proj = nn.Linear(n_embed, n_embed)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):

        out_list = []
        att_list = []

        for head in self.heads:
            out, att = head(x)
            out_list.append(out)
            att_list.append(att)

        out = torch.cat(out_list, dim=-1)  # Concatenate outputs from all heads
        out = self.proj(out)
        out = self.dropout(out)

        att_stack = torch.stack(att_list, dim=1) # stack attention matrices from all heads ( B, n_heads, T, T)

        return out, att_stack
```

- **Feed Forward Network:** A position-wise fully connected neural network applied independently to each position, adding non-linearity to the model.



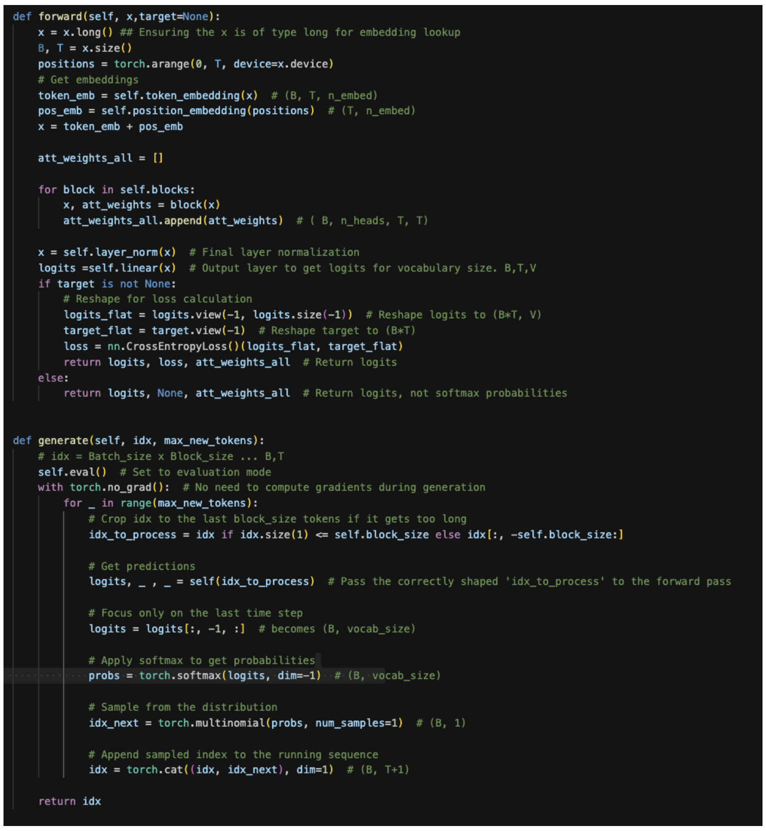
```python
class feed_forward(nn.Module):

    # Multi-layer perceptron (MLP) for feed-forward network in transformer

    def __init__(self, n_embed):
        super().__init__()
        self.network = nn.Sequential(
        nn.Linear(n_embed, 4 * n_embed),  # Up-projection min of 4* n_embed from the paper Attention Is All You Need
        nn.ReLU(),
        nn.Linear(4 * n_embed, n_embed),  # Down-projection back to n_embed
        nn.Dropout(dropout)
        )


    def forward(self, x):

        return self.network(x)
```

- **Block:** A single Transformer block consists of a Multi-Head Attention layer followed by a Feed-Forward Network, with residual connections and LayerNormalization applied around each sub-layer. The full Transformer model is constructed by stacking several such blocks.



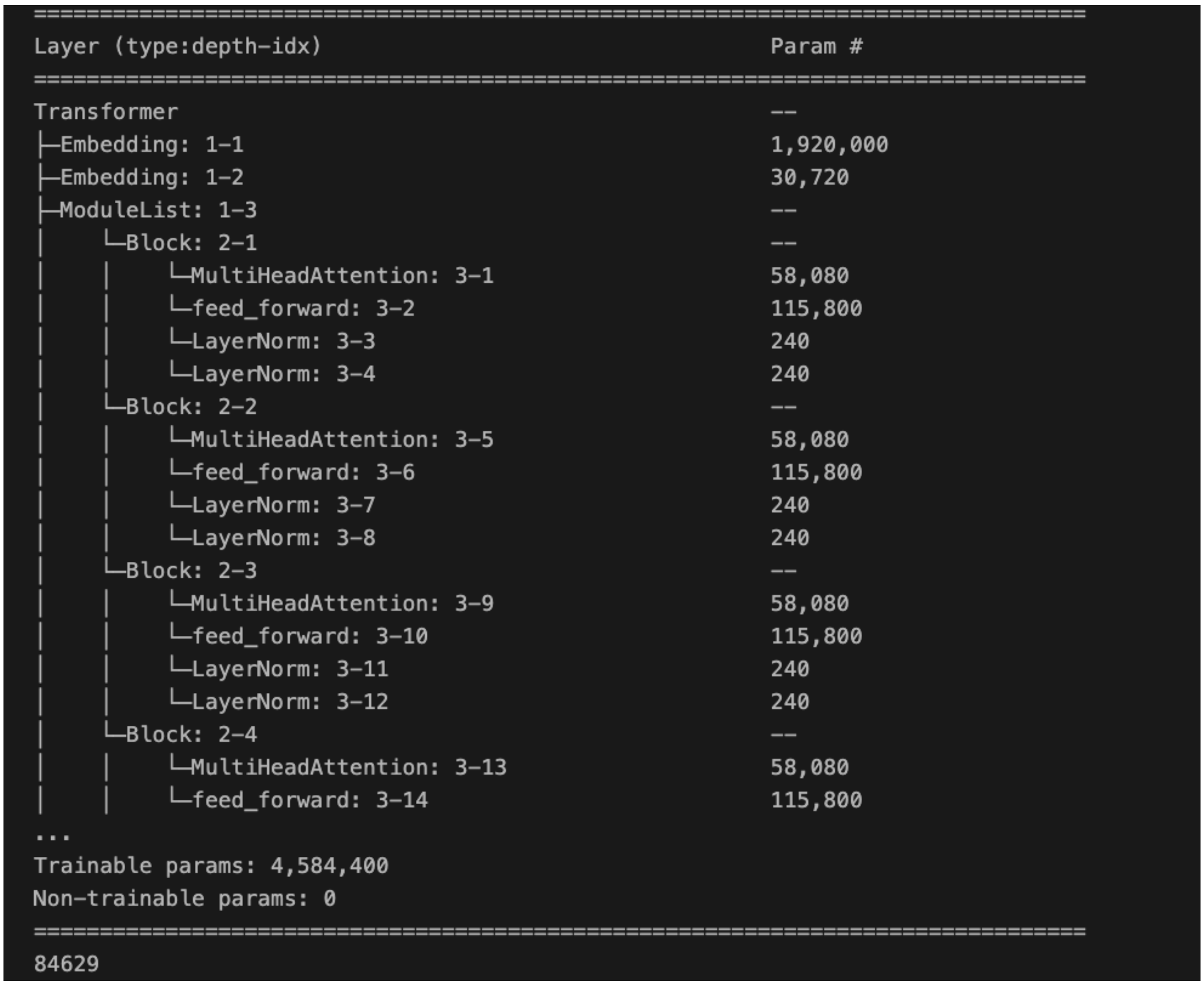
```python
#Single Bloack of the Transformer
class Block(nn.Module):

    def __init__(self, n_embed, n_heads):
        super().__init__()
        self.attention = MultiHeadAttention(n_embed, n_heads)
        self.feed_forward = feed_forward(n_embed)
        self.ln1 = nn.LayerNorm(n_embed)
        self.ln2 = nn.LayerNorm(n_embed)

    def forward(self, x):

        # Creating a residual connection around the attention layer
        norm_x = self.ln1(x)
        att_out, att_weights = self.attention(norm_x)

        x = x + att_out
        x = x + self.feed_forward(self.ln2(x))
        return x, att_weights
```

- **Generation:** The model's final layer predicts the probability distribution over the vocabulary for the next token. During inference, this allows for auto-regressive text generation, where the predicted token is appended to the input, and the process repeats to generate a continuous sequence.



```python
def forward(self, x,target=None):
    x = x.long() ## Ensuring the x is of type long for embedding lookup
    B, T = x.size()
    positions = torch.arange(0, T, device=x.device)
    # Get embeddings
    token_emb = self.token_embedding(x)  # (B, T, n_embed)
    pos_emb = self.position_embedding(positions)  # (T, n_embed)
    x = token_emb + pos_emb

    att_weights_all = []

    for block in self.blocks:
        x, att_weights = block(x)
        att_weights_all.append(att_weights)  # ( B, n_heads, T, T)

    x = self.layer_norm(x)  # Final layer normalization
    logits =self.linear(x)  # Output layer to get logits for vocabulary size. B,T,V
    if target is not None:
        # Reshape for loss calculation
        logits_flat = logits.view(-1, logits.size(-1))  # Reshape logits to (B*T, V)
        target_flat = target.view(-1)  # Reshape target to (B*T)
        loss = nn.CrossEntropyLoss()(logits_flat, target_flat)
        return logits, loss, att_weights_all  # Return logits
    else:
        return logits, None, att_weights_all  # Return logits, not softmax probabilities


def generate(self, idx, max_new_tokens):
    # idx = Batch_size x Block_size ... B,T
    self.eval()  # Set to evaluation mode
    with torch.no_grad():  # No need to compute gradients during generation
        for _ in range(max_new_tokens):
            # Crop idx to the last block_size tokens if it gets too long
            idx_to_process = idx if idx.size(1) <= self.block_size else idx[:, -self.block_size:]

            # Get predictions
            logits, _ , _ = self(idx_to_process)  # Pass the correctly shaped 'idx_to_process' to the forward pass

            # Focus only on the last time step
            logits = logits[:, -1, :]  # becomes (B, vocab_size)

            # Apply softmax to get probabilities
            probs = torch.softmax(logits, dim=-1)  # (B, vocab_size)

            # Sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)  # (B, 1)

            # Append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1)  # (B, T+1)

    return idx
```

The resultant model has a total of **4.5 million parameters** which is trained on the total length of **1692913 (1.6 Million) token_ids** which rounded up to **84629 batch per epoch**. The final structure of the model is given below.

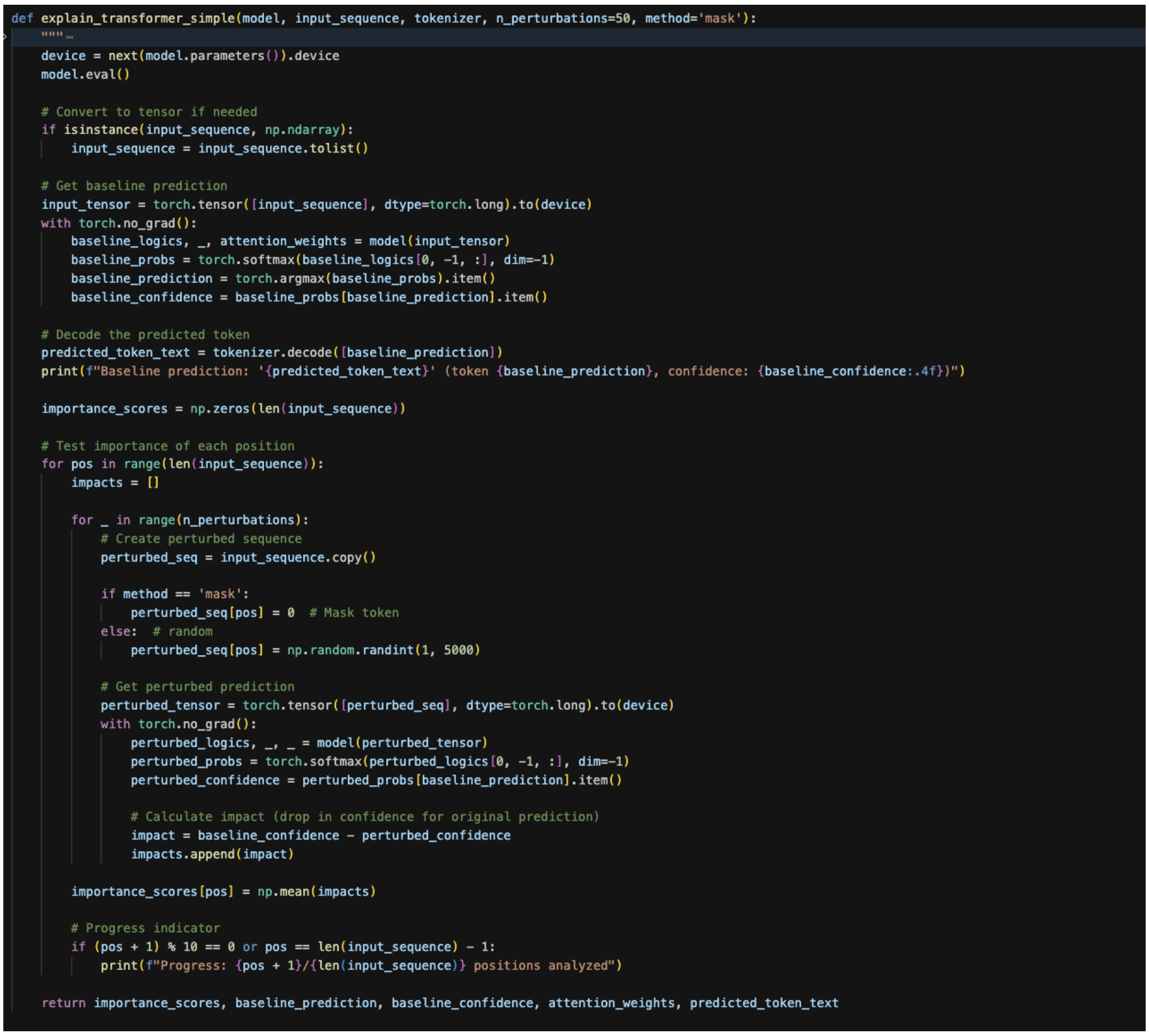


```
================================================================================
Layer (type:depth-idx)                                        Param #
================================================================================
Transformer                                                   --
├─Embedding: 1-1                                              1,920,000
├─Embedding: 1-2                                              30,720
├─ModuleList: 1-3                                             --
│    └─Block: 2-1                                             --
│    │    └─MultiHeadAttention: 3-1                           58,080
│    │    └─feed_forward: 3-2                                 115,800
│    │    └─LayerNorm: 3-3                                    240
│    │    └─LayerNorm: 3-4                                    240
│    └─Block: 2-2                                             --
│    │    └─MultiHeadAttention: 3-5                           58,080
│    │    └─feed_forward: 3-6                                 115,800
│    │    └─LayerNorm: 3-7                                    240
│    │    └─LayerNorm: 3-8                                    240
│    └─Block: 2-3                                             --
│    │    └─MultiHeadAttention: 3-9                           58,080
│    │    └─feed_forward: 3-10                                115,800
│    │    └─LayerNorm: 3-11                                   240
│    │    └─LayerNorm: 3-12                                   240
│    └─Block: 2-4                                             --
│    │    └─MultiHeadAttention: 3-13                          58,080
│    │    └─feed_forward: 3-14                                115,800
...
Trainable params: 4,584,400
Non-trainable params: 0
================================================================================
84629
```

5.5 Explainable AI Integration

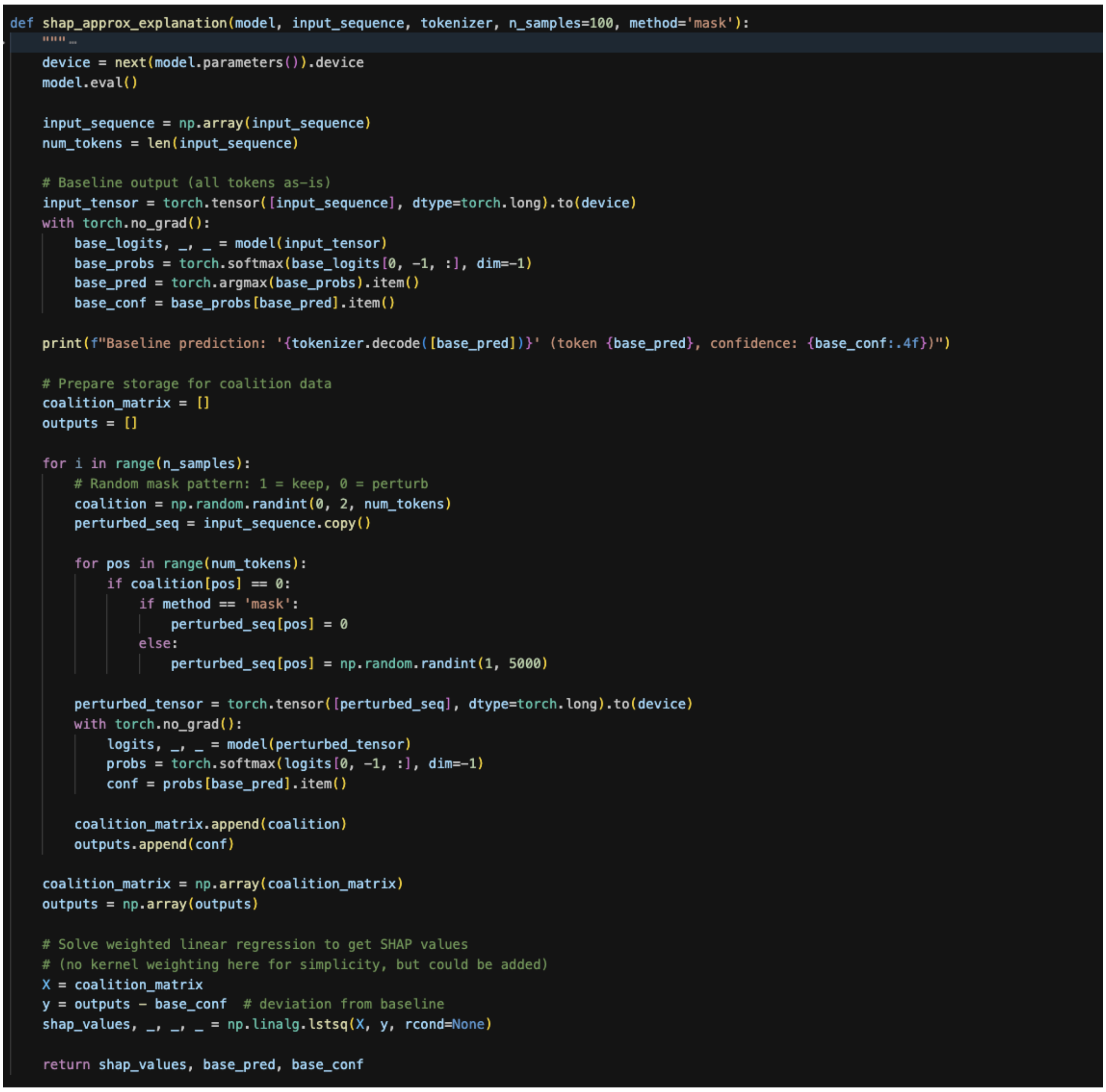
The project integrates XAI directly after model training for post-hoc interpretability:

- **Perturbation-Based Feature Attribution:** This method is implemented to calculate the importance of each input token by observing the drop in prediction confidence when that token is masked or randomized.



```python
def explain_transformer_simple(model, input_sequence, tokenizer, n_perturbations=50, method='mask'):
    """…
    device = next(model.parameters()).device
    model.eval()

    # Convert to tensor if needed
    if isinstance(input_sequence, np.ndarray):
        input_sequence = input_sequence.tolist()

    # Get baseline prediction
    input_tensor = torch.tensor([input_sequence], dtype=torch.long).to(device)
    with torch.no_grad():
        baseline_logics, _, attention_weights = model(input_tensor)
        baseline_probs = torch.softmax(baseline_logics[0, -1, :], dim=-1)
        baseline_prediction = torch.argmax(baseline_probs).item()
        baseline_confidence = baseline_probs[baseline_prediction].item()

    # Decode the predicted token
    predicted_token_text = tokenizer.decode([baseline_prediction])
    print(f"Baseline prediction: '{predicted_token_text}' (token {baseline_prediction}, confidence: {baseline_confidence:.4f})")

    importance_scores = np.zeros(len(input_sequence))

    # Test importance of each position
    for pos in range(len(input_sequence)):
        impacts = []

        for _ in range(n_perturbations):
            # Create perturbed sequence
            perturbed_seq = input_sequence.copy()

            if method == 'mask':
                perturbed_seq[pos] = 0  # Mask token
            else:  # random
                perturbed_seq[pos] = np.random.randint(1, 5000)

            # Get perturbed prediction
            perturbed_tensor = torch.tensor([perturbed_seq], dtype=torch.long).to(device)
            with torch.no_grad():
                perturbed_logics, _, _ = model(perturbed_tensor)
                perturbed_probs = torch.softmax(perturbed_logics[0, -1, :], dim=-1)
                perturbed_confidence = perturbed_probs[baseline_prediction].item()

                # Calculate impact (drop in confidence for original prediction)
                impact = baseline_confidence - perturbed_confidence
                impacts.append(impact)

        importance_scores[pos] = np.mean(impacts)

        # Progress indicator
        if (pos + 1) % 10 == 0 or pos == len(input_sequence) - 1:
            print(f"Progress: {pos + 1}/{len(input_sequence)} positions analyzed")

    return importance_scores, baseline_prediction, baseline_confidence, attention_weights, predicted_token_text
```

- **SHAP Approximation (Linear SHAP):** An approximate SHAP implementation is utilized. This involves sampling various token coalitions, perturbing the omitted tokens, measuring the change in prediction confidence, and then fitting a linear model to estimate the contribution (Shapley value) of each token.



```python
def shap_approx_explanation(model, input_sequence, tokenizer, n_samples=100, method='mask'):
    """…
    device = next(model.parameters()).device
    model.eval()

    input_sequence = np.array(input_sequence)
    num_tokens = len(input_sequence)

    # Baseline output (all tokens as-is)
    input_tensor = torch.tensor([input_sequence], dtype=torch.long).to(device)
    with torch.no_grad():
        base_logits, _, _ = model(input_tensor)
        base_probs = torch.softmax(base_logits[0, -1, :], dim=-1)
        base_pred = torch.argmax(base_probs).item()
        base_conf = base_probs[base_pred].item()

    print(f"Baseline prediction: '{tokenizer.decode([base_pred])}' (token {base_pred}, confidence: {base_conf:.4f})")

    # Prepare storage for coalition data
    coalition_matrix = []
    outputs = []

    for i in range(n_samples):
        # Random mask pattern: 1 = keep, 0 = perturb
        coalition = np.random.randint(0, 2, num_tokens)
        perturbed_seq = input_sequence.copy()

        for pos in range(num_tokens):
            if coalition[pos] == 0:
                if method == 'mask':
                    perturbed_seq[pos] = 0
                else:
                    perturbed_seq[pos] = np.random.randint(1, 5000)

        perturbed_tensor = torch.tensor([perturbed_seq], dtype=torch.long).to(device)
        with torch.no_grad():
            logits, _, _ = model(perturbed_tensor)
            probs = torch.softmax(logits[0, -1, :], dim=-1)
            conf = probs[base_pred].item()

        coalition_matrix.append(coalition)
        outputs.append(conf)

    coalition_matrix = np.array(coalition_matrix)
    outputs = np.array(outputs)

    # Solve weighted linear regression to get SHAP values
    # (no kernel weighting here for simplicity, but could be added)
    X = coalition_matrix
    y = outputs - base_conf  # deviation from baseline
    shap_values, _, _, _ = np.linalg.lstsq(X, y, rcond=None)

    return shap_values, base_pred, base_conf
```

Both XAI methods generate importance scores, which are then used to visualize the influence of each input token on the model's next-word prediction.

6. Results and Discussion

The trained Transformer model demonstrates its capability in predicting subsequent tokens given a context. For instance, providing the model with a partial sentence allows it to complete the thought based on the patterns learned from the diverse training data.

6.1 Analysing the Text Generation

Example 1 (Poetry):

- **Input Sequence:**
  "Ah! mayst thou ever be what now thou art,
  Nor unbeseem the promise of thy spring,
  As fair in form, as warm yet pure in heart,
  Love's image upon earth without his wing,"
- **Model Predicted :**

  "Ah! mayst thou ever be what now thou art,

  Nor unbeseem the promise of thy spring,

  As fair in form, as warm yet pure in heart,

  Love's image upon earth without his wing,

  and neither it was sufficient for a mountain;

  nor beaten up, leaving the horses wrong. "

Example 2 (Literature):

- **Input Sequence:** "A HOSPITAL assistant, called Yergunov, an empty-headed fellow, known throughout the district as a great braggart and drunkard, was returning one evening in Christmas week from"
- **Model Predicted :** "A HOSPITAL assistant, called Yergunov, an empty-headed fellow, known throughout the district as a great braggart and drunkard, was returning one evening in Christmas week from the rural rhyme. I had been married, a terrible appetite, the second forty-five "

Example 3 (Academic)::

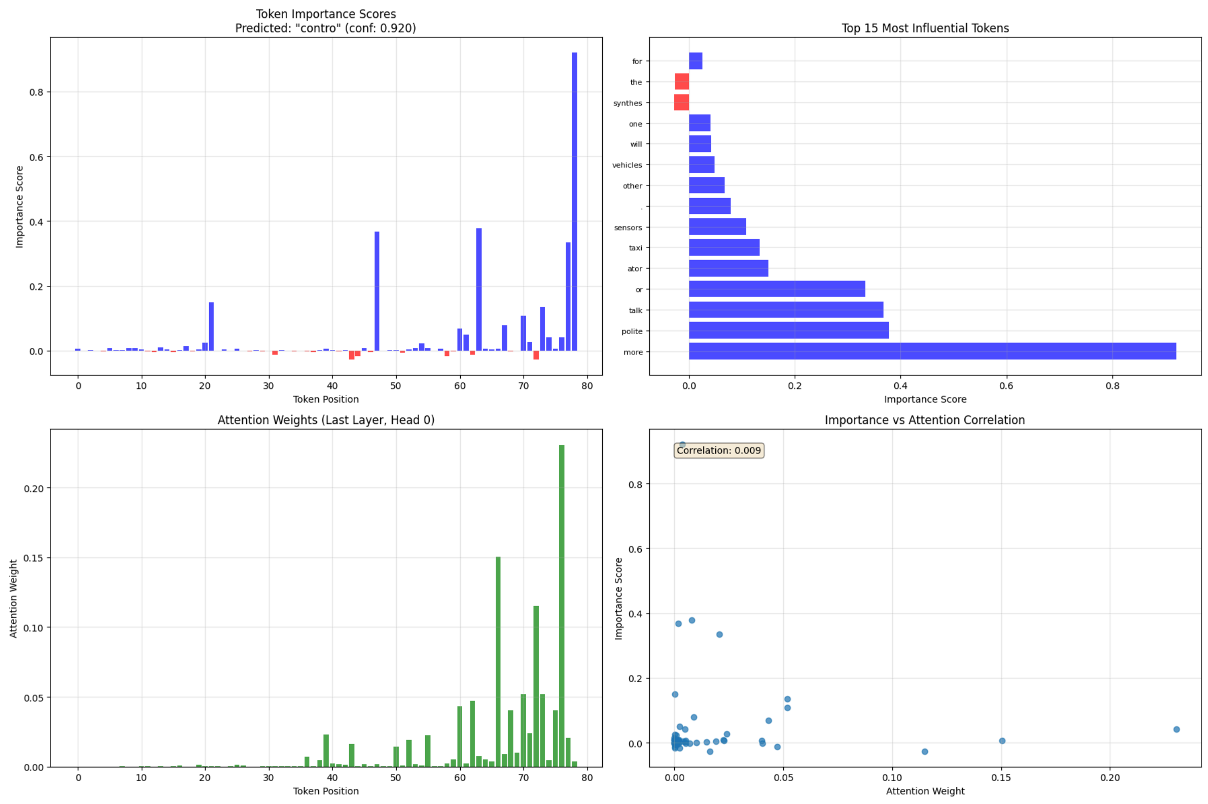
- **Input Sequence:** “Here, we compare the theoretical predictions of α-attractor T-model modified by GUP with recent observational data to assess”
- **Model Predicted :** “Here, we compare the theoretical predictions of α-attractor T-model modified by GUP with recent observational data to assess the combined data are consistent with the upper version and consistent empirical findings on traditional relationship between higher GUP as ”

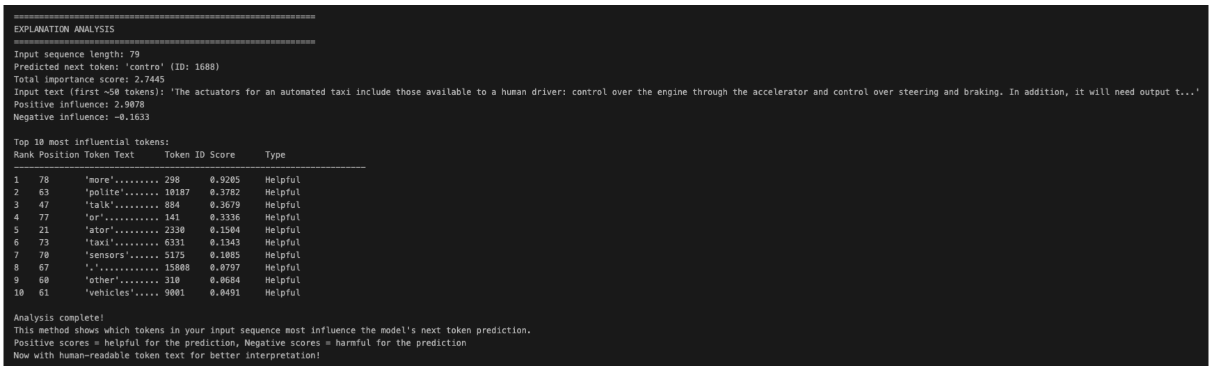
When we apply our XAI methods (Perturbation-Based Attribution and SHAP) to this prediction, we obtain importance scores for each token in the input sequence. These scores are then visualized, often as heatmaps or bar charts, to highlight influential tokens.

6.2 Visualization in Perturbation-Based Attribution

Example:

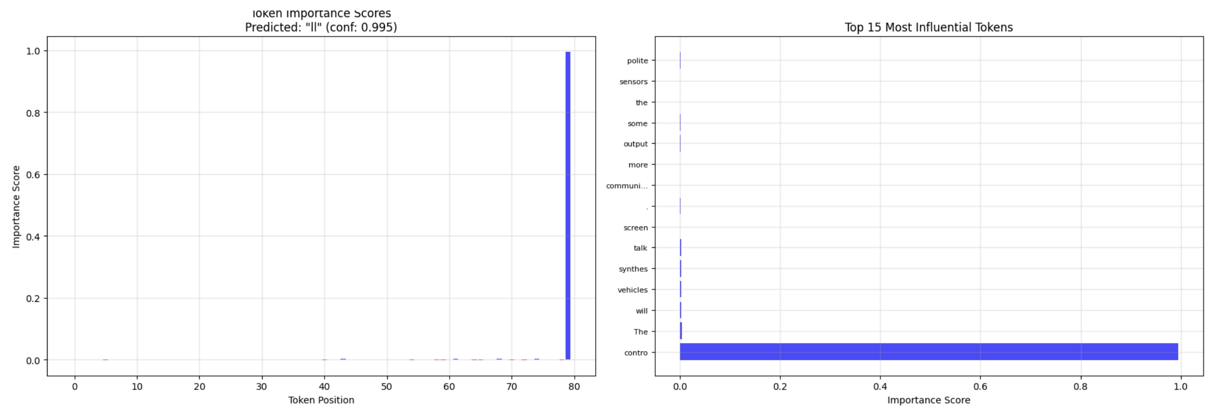
- **Input Sequence:** “The actuators for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise. The basic sensors for the taxi will include one or more”
- **Model Predicted :** “contro ”



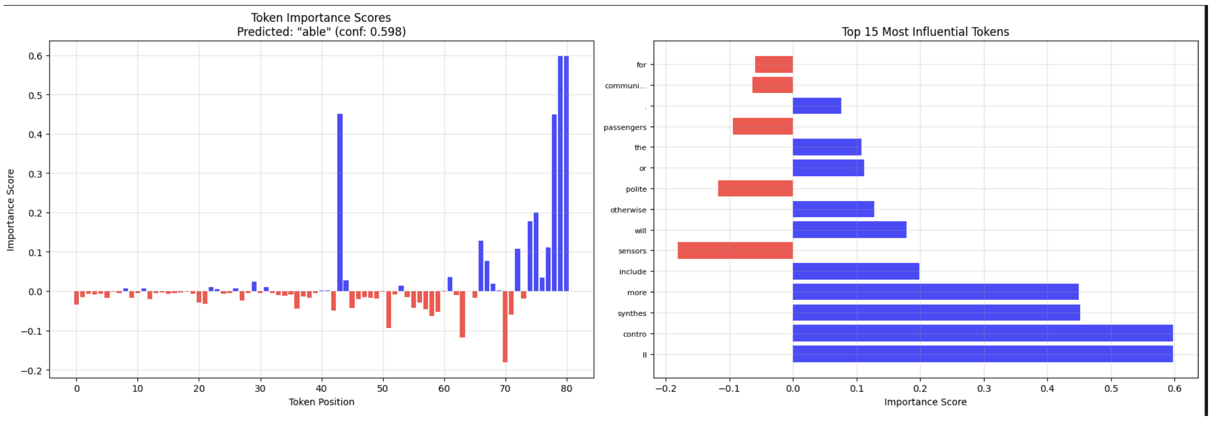


```
============================================================
EXPLANATION ANALYSIS
============================================================
Input sequence length: 79
Predicted next token: 'contro' (ID: 1688)
Total importance score: 2.7445
Input text (first ~50 tokens): 'The actuators for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output t...'
Positive influence: 2.9078
Negative influence: -0.1633

Top 10 most influential tokens:
Rank Position Token Text      Token ID Score      Type
----------------------------------------------------------------------
1    78       'more'......... 298      0.9205     Helpful
2    63       'polite'....... 10187    0.3782     Helpful
3    47       'talk'......... 884      0.3679     Helpful
4    77       'or'........... 141      0.3336     Helpful
5    21       'ator'......... 2330     0.1504     Helpful
6    73       'taxi'......... 6331     0.1343     Helpful
7    70       'sensors'...... 5175     0.1085     Helpful
8    67       '.'............ 15808    0.0797     Helpful
9    60       'other'........ 310      0.0684     Helpful
10   61       'vehicles'..... 9001     0.0491     Helpful

Analysis complete!
This method shows which tokens in your input sequence most influence the model's next token prediction.
Positive scores = helpful for the prediction, Negative scores = harmful for the prediction
Now with human-readable token text for better interpretation!
```

When the “contro” was appended in the input sequence the prediction was ‘ll’ which was highly influenced by ‘contro’



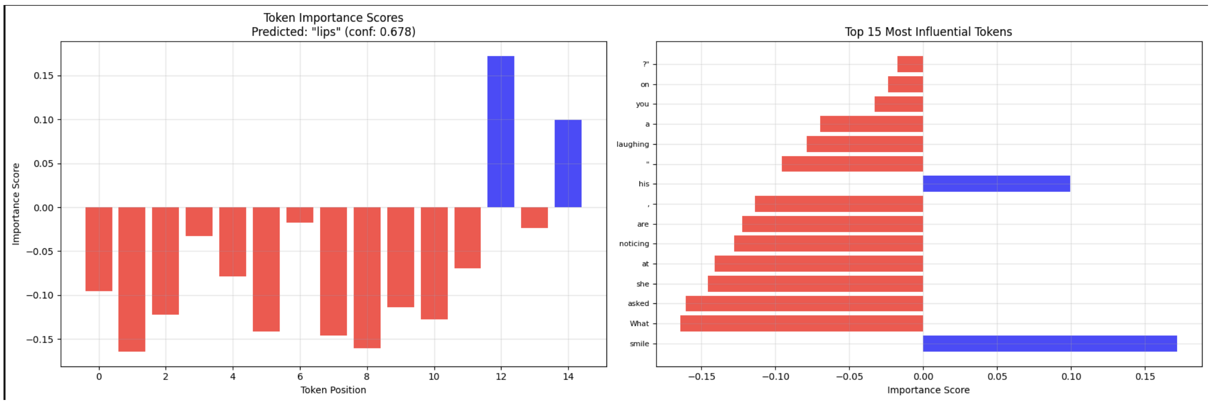
Similarly appending “ll” again gave “able”

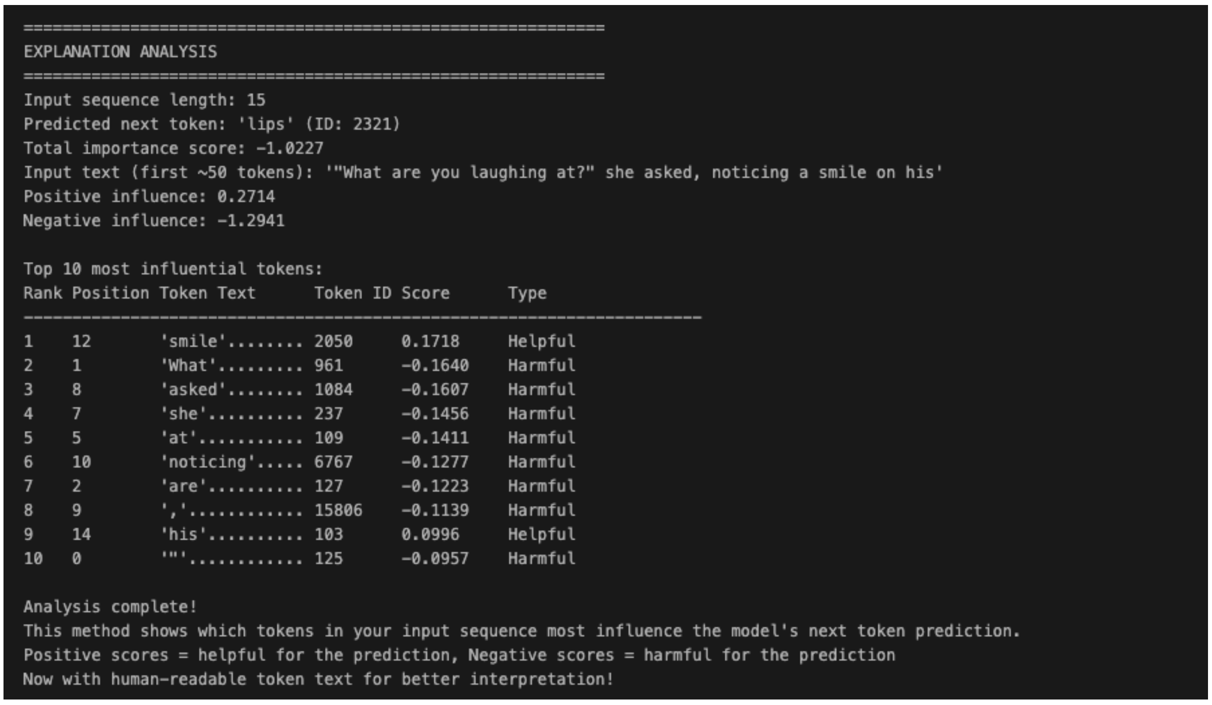


6.3 Visualization in LinearSHAP

Example:

- **Input Sequence:** “What are you laughing at?" she asked, noticing a smile on his”
- **Model Predicted :** “lips ”



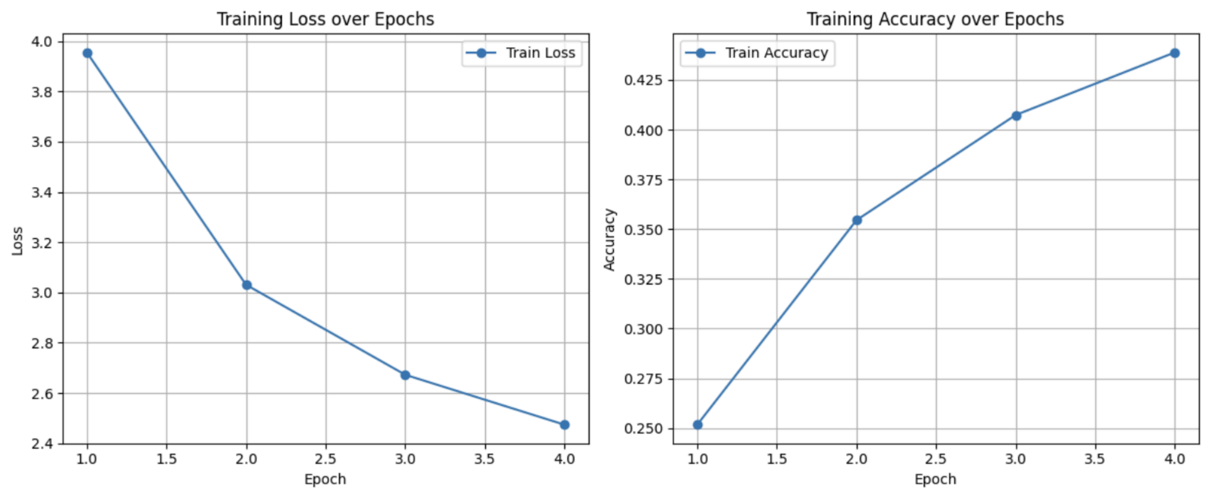


```
============================================================
EXPLANATION ANALYSIS
============================================================
Input sequence length: 15
Predicted next token: 'lips' (ID: 2321)
Total importance score: -1.0227
Input text (first ~50 tokens): '"What are you laughing at?" she asked, noticing a smile on his'
Positive influence: 0.2714
Negative influence: -1.2941

Top 10 most influential tokens:
Rank Position Token Text      Token ID Score      Type
----------------------------------------------------------------------
1    12       'smile'........ 2050     0.1718     Helpful
2    1        'What'......... 961      -0.1640    Harmful
3    8        'asked'........ 1084     -0.1607    Harmful
4    7        'she'.......... 237      -0.1456    Harmful
5    5        'at'........... 109      -0.1411    Harmful
6    10       'noticing'..... 6767     -0.1277    Harmful
7    2        'are'.......... 127      -0.1223    Harmful
8    9        ','............ 15806    -0.1139    Harmful
9    14       'his'.......... 103      0.0996     Helpful
10   0        '"'............ 125      -0.0957    Harmful

Analysis complete!
This method shows which tokens in your input sequence most influence the model's next token prediction.
Positive scores = helpful for the prediction, Negative scores = harmful for the prediction
Now with human-readable token text for better interpretation!
```

These visualizations provide crucial insights:

- **Transparency:** They reveal *why* the model made a particular prediction, moving beyond a simple output to show the underlying reasoning.
- **Debugging:** If the model makes an incorrect prediction, XAI can help identify if it focused on irrelevant tokens or missed important ones.
- **Trust:** By showing the influential factors, users can develop greater trust in the model, especially in high-stakes applications.

Also the figure below shows the graph between training loss and training accuracies.



7. Conclusion

This mini-project successfully achieved its dual objectives: building a Transformer-based next-word predictor entirely from scratch and integrating advanced Explainable AI techniques. By custom-implementing the Transformer architecture, we gained a deep understanding of its components, from self-attention to multi-head attention and stacked blocks. Furthermore, the application of perturbation-based feature attribution and approximate SHAP provided invaluable insights into the model's decision-making process, allowing us to identify and visualize the specific tokens that most influenced its next-word predictions. This work highlights the critical importance of interpretability in AI models, particularly as they become more complex and deployed in real-world scenarios.

References

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems, 30.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training. OpenAI.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers).

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. arXiv preprint arXiv:2001.08361.

Subramanian, S., Elango, V., & Gungor, M. (2025). Small Language Models (SLMs) Can Still Pack a Punch: A survey. arXiv preprint arXiv:2501.05465.

Zhao, Y., Li, S., Wang, J., & Chen, Y. (2025). Small Language Models in the Real World: Insights from Industrial Text Classification. [Placeholder for actual publication details, based on search results].

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Lundberg, S. M., & Lee, S. I. (2017). A Unified Approach to Interpreting Model Predictions. Advances in Neural Information Processing Systems, 30.

Sundararajan, M., Taly, A., & Yan, Q. (2017). Axiomatic Attribution for Deep Networks. Proceedings of the 34th International Conference on Machine Learning.

Doshi-Velez, F., & Kim, B. (2017). Towards a rigorous science of interpretable machine learning. arXiv preprint arXiv:1702.08608. (Note: This is a general XAI survey, not specifically perturbation-based, but relevant to the broader context of XAI.)

Project Link: https://github.com/sidartchy/language_modeling_transformer_xai