
**Bangladesh University of Professionals (BUP)**

**Faculty of Science & Technology (FST)**

**Department of Computer Science and Engineering**

**Master's in computer science and engineering (MCSE) Professional Program**

# ASSIGNMENT

**Name of The Topic :** Transformer Tutorial (Application Based)

**Course Title :** Natural Language Processing

**Course Code :** MCSE 2202

**Submitted To :** **Dr. Fazlul Hasan Siddiqui**
Professor,
Department of Computer Science and Engineering
DUET

**Submitted By :** Saad Bin Hasan Sakib
ID No: 24525002007
2nd Batch (MCSE)

**Date of Submission:** **14-11-2025**

## Table of Contents

## Abstract

This report presents a complete beginner-friendly guide to understanding and implementing a Transformer model from scratch using PyTorch. The objective is to simplify the complex ideas behind Transformers so even someone with no prior deep learning experience can follow along. We build a tiny character-level language model and explain every step: tokenization, dataset preparation, model architecture, multi-head attention, positional encoding, training, and text generation.

## 1. Introduction

Transformers are currently the backbone of modern artificial intelligence. ChatGPT, GPT-4, BERT, and almost every high-performing NLP model today uses the Transformer architecture. However, most beginners struggle to understand how Transformers work under the hood. This mini-book/report explains everything from scratch with code, diagrams, and real examples.

## 2. Background

### 2.1 Why Language Models?

A language model predicts the next word or next character given previous text. Examples:
- Autocomplete in mobile keyboards
- Search engine suggestions
- Chatbots
- Text generation

### 2.2 Problems with RNNs/LSTMs

Before Transformers, RNNs and LSTMs were used, but they had issues:
- Slow training due to sequential processing
- Forget long-term dependencies
- Hard to parallelize

Transformers solve these using self-attention.

## 3. Understanding Self-Attention

Self-attention answers a simple question:
**'When reading this token, which other tokens should I pay attention to? '**

For example, in:
“The cat sat on the mat because it was tired.”
The word **'it'** refers to **'cat'** — attention helps the model learn such connections.

## 4. Building a Tiny Transformer From Scratch

We now build a decoder-only Transformer (like a mini GPT). The full system includes:
- Tokenizer (character-level)
- Dataset builder
- Embeddings
- Positional encoding
- Multi-head self-attention
- Feed-forward block
- Stacked transformer layers
- Autoregressive text generation

## 5. Full Code With Step-by-Step Explanation

### 5.1 data.py — Tokenizer, Dataset, and DataLoader

This file prepares text, tokenizes characters, and creates training sequences.

```
import os
from typing import Tuple, Dict

import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader


class CharTokenizer:
    """
    Simple character-level tokenizer:
    - builds vocab from text
    - encodes text -> list[int]
    - decodes list[int] -> text
    """
    def __init__(self, text: str):
        chars = sorted(list(set(text)))
        self.vocab_size = len(chars)
        self.stoi: Dict[str, int] = {ch: i for i, ch in enumerate(chars)}
        self.itos: Dict[int, str] = {i: ch for ch, i in self.stoi.items()}

    def encode(self, s: str):
        """String -> list of ids"""
        return [self.stoi[c] for c in s]

    def decode(self, ids):
        """List of ids -> string"""
        return ''.join([self.itos[i] for i in ids])


class CharDataset(Dataset):
```

```
    """
    Dataset of overlapping sequences for next-token prediction.
    Given a long sequence of token ids, we create pairs:
        x = ids[i : i+block_size]
        y = ids[i+1 : i+1+block_size]
    """
    def __init__(self, data: torch.Tensor, block_size: int):
        super().__init__()
        self.data = data
        self.block_size = block_size

    def __len__(self):
        # len(data) - block_size হতে যদি negative আসে,
        # সেটাকে 0 করে দিচ্ছি যাতে DataLoader error না দেয়।
        return max(0, len(self.data) - self.block_size)

    def __getitem__(self, idx):
        x = self.data[idx : idx + self.block_size]
        y = self.data[idx + 1 : idx + 1 + self.block_size]
        return x, y


def load_data(
    corpus_path: str,
    block_size: int = 64,
    batch_size: int = 32,
    train_split: float = 0.8,
) -> Tuple[DataLoader, DataLoader, CharTokenizer]:
    """
    Reads corpus.txt, builds tokenizer, creates train & val DataLoaders.

    Returns:
        train_loader, val_loader, tokenizer
    """
    assert os.path.isfile(corpus_path), f"Corpus file not found: {corpus_path}"

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

    # build tokenizer
    tokenizer = CharTokenizer(text)
    print(f"Loaded corpus with {len(text)} characters, vocab size =
{tokenizer.vocab_size}")

    # encode entire text
    data_ids = torch.tensor(tokenizer.encode(text), dtype=torch.long)

    # train/val split
    n = int(len(data_ids) * train_split)
```

```
    train_data = data_ids[:n]
    val_data = data_ids[n:]

    train_dataset = CharDataset(train_data, block_size)
    val_dataset = CharDataset(val_data, block_size)

    train_loader = DataLoader(
        train_dataset,
        batch_size=batch_size,
        shuffle=True,
        drop_last=True,    # train এ small last batch ফেলে দিচ্ছি
    )
    val_loader = DataLoader(
        val_dataset,
        batch_size=batch_size,
        shuffle=False,
        drop_last=False,   # validation এ small batch retain করছি
    )

    return train_loader, val_loader, tokenizer
```

### 5.2 model.py — Tiny Transformer Implementation

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


class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, d_model: int):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        # x: (batch, seq_len)
        return self.embedding(x)  # (batch, seq_len, d_model)


class PositionalEmbedding(nn.Module):
    """
    Learnable positional embedding: position index -> vector
    """
    def __init__(self, max_len: int, d_model: int):
        super().__init__()
        self.embedding = nn.Embedding(max_len, d_model)

    def forward(self, x):
```

```
        # x: (batch, seq_len)
        b, seq_len = x.size()
        positions = torch.arange(0, seq_len, device=x.device).unsqueeze(0)  # (1,
seq_len)
        return self.embedding(positions)  # (1, seq_len, d_model)


class MultiHeadSelfAttention(nn.Module):
    def __init__(self, d_model: int, num_heads: int):
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_head = d_model // num_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)

    def forward(self, x, mask=None):
        # x: (batch, seq_len, d_model)
        b, seq_len, _ = x.size()

        # 1. Linear projections
        Q = self.W_q(x)  # (b, seq_len, d_model)
        K = self.W_k(x)
        V = self.W_v(x)

        # 2. Split into heads
        # (b, seq_len, num_heads, d_head) -> (b, num_heads, seq_len, d_head)
        Q = Q.view(b, seq_len, self.num_heads, self.d_head).transpose(1, 2)
        K = K.view(b, seq_len, self.num_heads, self.d_head).transpose(1, 2)
        V = V.view(b, seq_len, self.num_heads, self.d_head).transpose(1, 2)

        # 3. Scaled dot-product attention
        # scores: (b, num_heads, seq_len, seq_len)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_head)

        if mask is not None:
            # mask expected shape: (1, 1, seq_len, seq_len)
            scores = scores.masked_fill(mask == 0, float('-inf'))

        attn = F.softmax(scores, dim=-1)
        out = torch.matmul(attn, V)  # (b, num_heads, seq_len, d_head)

        # 4. Concatenate heads
        out = out.transpose(1, 2).contiguous().view(b, seq_len, self.d_model)
```

```
        # 5. Final linear
        out = self.W_o(out)
        return out


class FeedForward(nn.Module):
    def __init__(self, d_model: int, d_ff: int, dropout: float = 0.1):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )

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


class TransformerBlock(nn.Module):
    def __init__(self, d_model: int, num_heads: int, d_ff: int, dropout: float =
0.1):
        super().__init__()
        self.attn = MultiHeadSelfAttention(d_model, num_heads)
        self.ln1 = nn.LayerNorm(d_model)
        self.ff = FeedForward(d_model, d_ff, dropout)
        self.ln2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention + residual + layernorm
        attn_out = self.attn(x, mask)
        x = self.ln1(x + self.dropout(attn_out))

        # Feed-forward + residual + layernorm
        ff_out = self.ff(x)
        x = self.ln2(x + self.dropout(ff_out))
        return x


class TinyTransformerLM(nn.Module):
    """
    Tiny decoder-only Transformer Language Model
    """
    def __init__(
        self,
        vocab_size: int,
        d_model: int = 128,
        num_heads: int = 4,
```

```
        num_layers: int = 2,
        d_ff: int = 256,
        max_seq_len: int = 128,
        dropout: float = 0.1,
    ):
        super().__init__()
        self.vocab_size = vocab_size
        self.d_model = d_model
        self.max_seq_len = max_seq_len

        self.token_emb = TokenEmbedding(vocab_size, d_model)
        self.pos_emb = PositionalEmbedding(max_seq_len, d_model)
        self.dropout = nn.Dropout(dropout)

        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])

        self.ln_final = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size, bias=False)

    def _generate_causal_mask(self, seq_len: int, device):
        # lower-triangular matrix (1: allowed, 0: masked)
        mask = torch.tril(torch.ones(seq_len, seq_len,
device=device)).unsqueeze(0).unsqueeze(0)
        return mask  # (1, 1, seq_len, seq_len)

    def forward(self, x):
        # x: (batch, seq_len)
        b, seq_len = x.size()
        if seq_len > self.max_seq_len:
            raise ValueError(f"seq_len={seq_len} >
max_seq_len={self.max_seq_len}")

        token_emb = self.token_emb(x)                # (b, seq_len, d_model)
        pos_emb = self.pos_emb(x)                    # (1, seq_len, d_model)
        h = token_emb + pos_emb                      # broadcasting
        h = self.dropout(h)

        mask = self._generate_causal_mask(seq_len, x.device)

        for block in self.blocks:
            h = block(h, mask=mask)

        h = self.ln_final(h)
        logits = self.head(h)                        # (b, seq_len, vocab_size)
        return logits
```

```
    @torch.no_grad()
    def generate(self, idx, max_new_tokens: int):
        """
        Autoregressive generation:
        idx: (1, current_seq_len) start tokens
        """
        for _ in range(max_new_tokens):
            # crop to max_seq_len
            idx_cond = idx[:, -self.max_seq_len:]

            # forward
            logits = self(idx_cond)  # (1, seq_len, vocab_size)

            # last time step
            logits_last = logits[:, -1, :]  # (1, vocab_size)

            # softmax -> probabilities
            probs = F.softmax(logits_last, dim=-1)

            # sample next token (or use argmax)
            next_token = torch.multinomial(probs, num_samples=1)  # (1, 1)

            # append
            idx = torch.cat([idx, next_token], dim=1)
        return idx
```

### 5.3 train.py — Training Loop and Generation

```
# src/train.py

import os
import torch
import torch.nn as nn
from torch.optim import Adam
from tqdm import tqdm
import matplotlib.pyplot as plt  # NEW: for plotting

from data import load_data
from model import TinyTransformerLM


def train(
    corpus_path: str = "../data/corpus.txt",
    block_size: int = 32,
    batch_size: int = 32,
    d_model: int = 128,
    num_heads: int = 4,
```

```
    num_layers: int = 2,
    d_ff: int = 256,
    max_seq_len: int = 64,
    dropout: float = 0.1,
    lr: float = 3e-4,
    epochs: int = 50,
    device: str = None,
    save_path: str = "../tiny_transformer.pt",
):
    if device is None:
        device = "cuda" if torch.cuda.is_available() else "cpu"
    print("Using device:", device)

    # 1. Load data (NOTE: train_split=0.8 so we get some validation data)
    train_loader, val_loader, tokenizer = load_data(
        corpus_path=corpus_path,
        block_size=block_size,
        batch_size=batch_size,
        train_split=0.8,  # changed from default 0.9
    )

    # 2. Build model
    model = TinyTransformerLM(
        vocab_size=tokenizer.vocab_size,
        d_model=d_model,
        num_heads=num_heads,
        num_layers=num_layers,
        d_ff=d_ff,
        max_seq_len=max_seq_len,
        dropout=dropout,
    ).to(device)

    print(model)
    print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

    # 3. Loss & optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=lr)

    # 4. For logging losses
    train_losses = []
    val_losses = []

    # 5. Training loop
    for epoch in range(1, epochs + 1):
        model.train()
        train_loss_sum = 0.0
        num_batches = 0
```

```
        pbar = tqdm(train_loader, desc=f"Epoch {epoch}/{epochs} [train]",
leave=False)
        for x, y in pbar:
            x = x.to(device)   # (batch, seq_len)
            y = y.to(device)   # (batch, seq_len)

            optimizer.zero_grad()

            logits = model(x)  # (batch, seq_len, vocab_size)

            # CrossEntropyLoss expects (N, C) and target (N)
            B, T, C = logits.shape
            loss = criterion(logits.view(B * T, C), y.view(B * T))

            loss.backward()
            optimizer.step()

            train_loss_sum += loss.item()
            num_batches += 1
            pbar.set_postfix(loss=loss.item())

        avg_train_loss = train_loss_sum / num_batches

        # Validation
        model.eval()
        val_loss_sum = 0.0
        val_batches = 0
        with torch.no_grad():
            for x, y in val_loader:
                x = x.to(device)
                y = y.to(device)

                logits = model(x)
                B, T, C = logits.shape
                vloss = criterion(logits.view(B * T, C), y.view(B * T))
                val_loss_sum += vloss.item()
                val_batches += 1

        avg_val_loss = val_loss_sum / max(1, val_batches)

        train_losses.append(avg_train_loss)
        val_losses.append(avg_val_loss)

        print(f"Epoch {epoch}: train_loss={avg_train_loss:.4f},
val_loss={avg_val_loss:.4f}")

    # 6. Save model & tokenizer info
    os.makedirs(os.path.dirname(save_path), exist_ok=True)
    torch.save({
```

```
        "model_state_dict": model.state_dict(),
        "vocab_size": tokenizer.vocab_size,
        "stoi": tokenizer.stoi,
        "itos": tokenizer.itos,
        "config": {
            "d_model": d_model,
            "num_heads": num_heads,
            "num_layers": num_layers,
            "d_ff": d_ff,
            "max_seq_len": max_seq_len,
            "dropout": dropout,
            "block_size": block_size,
        },
        "train_losses": train_losses,
        "val_losses": val_losses,
    }, save_path)
    print(f"Model saved to {save_path}")

    # 7. Plot loss curves code
    plt.figure()
    plt.plot(train_losses, label="Train loss")
    plt.plot(val_losses, label="Validation loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training and Validation Loss")
    plt.legend()
    plt.grid(True)

    # Save plot next to model file
    loss_plot_path = os.path.join(os.path.dirname(save_path), "loss_curve.png")
    plt.savefig(loss_plot_path, dpi=150, bbox_inches="tight")
    plt.close()
    print(f"Loss curve saved to {loss_plot_path}")

    # 8. Quick demo generation
    demo_prompt = "amar"
    print("\n=== Demo generation ===")
    print("Prompt:", demo_prompt)

    # encode prompt
    start_ids = [tokenizer.stoi[c] for c in demo_prompt if c in tokenizer.stoi]
    if not start_ids:
        print("Prompt characters not in vocab, skipping demo.")
        return

    idx = torch.tensor(start_ids, dtype=torch.long, device=device).unsqueeze(0)
    gen_ids = model.generate(idx, max_new_tokens=100)[0].tolist()
    generated_text = tokenizer.decode(gen_ids)
    print("Generated:", generated_text)
```

```
if __name__ == "__main__":
    train()
```

### 5.4 inference.py — Prompt-based Text generation

```
# src/inference.py

import sys
import argparse
import torch
import torch.nn.functional as F

from model import TinyTransformerLM


def load_model_and_tokenizer(checkpoint_path: str = "../tiny_transformer.pt",
device: str = None):
    """
    Load trained TinyTransformerLM and tokenizer info from checkpoint.
    """
    if device is None:
        device = "cuda" if torch.cuda.is_available() else "cpu"

    print(f"[INFO] Loading checkpoint from: {checkpoint_path}")
    ckpt = torch.load(checkpoint_path, map_location=device)

    vocab_size = ckpt["vocab_size"]
    stoi = ckpt["stoi"]
    itos = ckpt["itos"]
    config = ckpt["config"]

    model = TinyTransformerLM(
        vocab_size=vocab_size,
        d_model=config["d_model"],
        num_heads=config["num_heads"],
        num_layers=config["num_layers"],
        d_ff=config["d_ff"],
        max_seq_len=config["max_seq_len"],
        dropout=config["dropout"],
    ).to(device)

    model.load_state_dict(ckpt["model_state_dict"])
    model.eval()

    print(f"[INFO] Model loaded (vocab_size={vocab_size},
d_model={config['d_model']})")
```

```python
    return model, stoi, itos, device, config


def encode_prompt(prompt: str, stoi: dict):
    """
    Convert prompt string -> list of token ids.
    Unknown chars are skipped with a warning.
    """
    ids = []
    for ch in prompt:
        if ch not in stoi:
            print(f"[WARN] Character {repr(ch)} not in vocabulary. Skipping.")
            continue
        ids.append(stoi[ch])
    if not ids:
        raise ValueError("Prompt has no valid characters from the vocabulary.")
    return ids


def decode_ids(ids, itos: dict):
    """
    Convert list of token ids -> string.
    """
    return "".join(itos[i] for i in ids)


@torch.no_grad()
def sample_with_temperature(
    model: TinyTransformerLM,
    idx: torch.Tensor,
    max_new_tokens: int,
    device: str,
    max_seq_len: int,
    temperature: float = 1.0,
    top_k: int | None = None,
):
    """
    Custom autoregressive generation with:
    - temperature
    - top-k sampling

    Args:
        model: trained TinyTransformerLM
        idx: (1, current_seq_len) starting token ids
        max_new_tokens: how many new tokens to generate
        device: 'cpu' or 'cuda'
        max_seq_len: model's maximum sequence length
        temperature: >1.0 = more random, <1.0 = more greedy, 1.0 = default
        top_k: if not None, only keep top_k logits before softmax
    """
```

```
    model.eval()
    idx = idx.to(device)

    for _ in range(max_new_tokens):
        # crop context to last max_seq_len tokens
        idx_cond = idx[:, -max_seq_len:]

        # forward pass
        logits = model(idx_cond)  # (1, seq_len, vocab_size)

        # take last time step
        logits = logits[:, -1, :]  # (1, vocab_size)

        # apply temperature
        if temperature <= 0:
            raise ValueError("temperature must be > 0")
        logits = logits / temperature

        # optional top-k filtering
        if top_k is not None and top_k > 0:
            v, _ = torch.topk(logits, k=top_k, dim=-1)
            min_keep = v[:, -1].unsqueeze(-1)
            logits = torch.where(logits < min_keep, torch.full_like(logits,
float('-inf')), logits)

        # softmax -> probabilities
        probs = F.softmax(logits, dim=-1)  # (1, vocab_size)

        # sample from distribution
        next_token = torch.multinomial(probs, num_samples=1)  # (1,1)

        # append to sequence
        idx = torch.cat([idx, next_token], dim=1)  # (1, seq_len+1)

    return idx


def generate_text(
    prompt: str,
    max_new_tokens: int = 100,
    checkpoint_path: str = "../tiny_transformer.pt",
    temperature: float = 1.0,
    top_k: int | None = None,
    num_samples: int = 1,
):
    """
    High-level helper: load model, encode prompt, and generate multiple samples.
    """
```

```
    model, stoi, itos, device, config = load_model_and_tokenizer(checkpoint_path,
device=None)

    # encode prompt
    start_ids = encode_prompt(prompt, stoi)
    start_idx = torch.tensor(start_ids, dtype=torch.long,
device=device).unsqueeze(0)  # (1, seq_len)

    outputs = []
    for i in range(num_samples):
        print(f"\n[INFO] Generating sample {i+1}/{num_samples} ...")
        out_ids = sample_with_temperature(
            model=model,
            idx=start_idx.clone(),  # clone so each sample starts from same
prompt
            max_new_tokens=max_new_tokens,
            device=device,
            max_seq_len=config["max_seq_len"],
            temperature=temperature,
            top_k=top_k,
        )[0].tolist()

        text = decode_ids(out_ids, itos)
        outputs.append(text)

    return outputs


def parse_args():
    parser = argparse.ArgumentParser(
        description="Tiny Transformer LM Inference Script (with temperature &
top-k sampling)"
    )
    parser.add_argument(
        "prompt",
        type=str,
        nargs="?",
        default=None,
        help="Prompt text to start generation. If omitted, you will be asked
interactively.",
    )
    parser.add_argument(
        "--max_new_tokens",
        type=int,
        default=100,
        help="Number of new tokens to generate (default: 100).",
    )
    parser.add_argument(
        "--temperature",
```

```
        type=float,
        default=1.0,
        help="Sampling temperature >0 (default: 1.0). Higher = more random.",
    )
    parser.add_argument(
        "--top_k",
        type=int,
        default=0,
        help="If >0, use top-k sampling with this k (default: 0 = disabled).",
    )
    parser.add_argument(
        "--num_samples",
        type=int,
        default=1,
        help="How many samples to generate from the same prompt (default: 1).",
    )
    parser.add_argument(
        "--ckpt",
        type=str,
        default="../tiny_transformer.pt",
        help="Path to model checkpoint (default: ../tiny_transformer.pt).",
    )
    return parser.parse_args()


def main():
    args = parse_args()

    # If no prompt passed as CLI arg, ask interactively
    if args.prompt is None:
        prompt = input("Enter prompt text: ")
    else:
        prompt = args.prompt

    top_k = args.top_k if args.top_k > 0 else None

    print(f"\n[CONFIG]")
    print(f"  Prompt         : {repr(prompt)}")
    print(f"  Max new tokens : {args.max_new_tokens}")
    print(f"  Temperature    : {args.temperature}")
    print(f"  Top-k          : {top_k}")
    print(f"  Num samples    : {args.num_samples}")
    print(f"  Checkpoint     : {args.ckpt}")

    outputs = generate_text(
        prompt=prompt,
        max_new_tokens=args.max_new_tokens,
        checkpoint_path=args.ckpt,
        temperature=args.temperature,
```

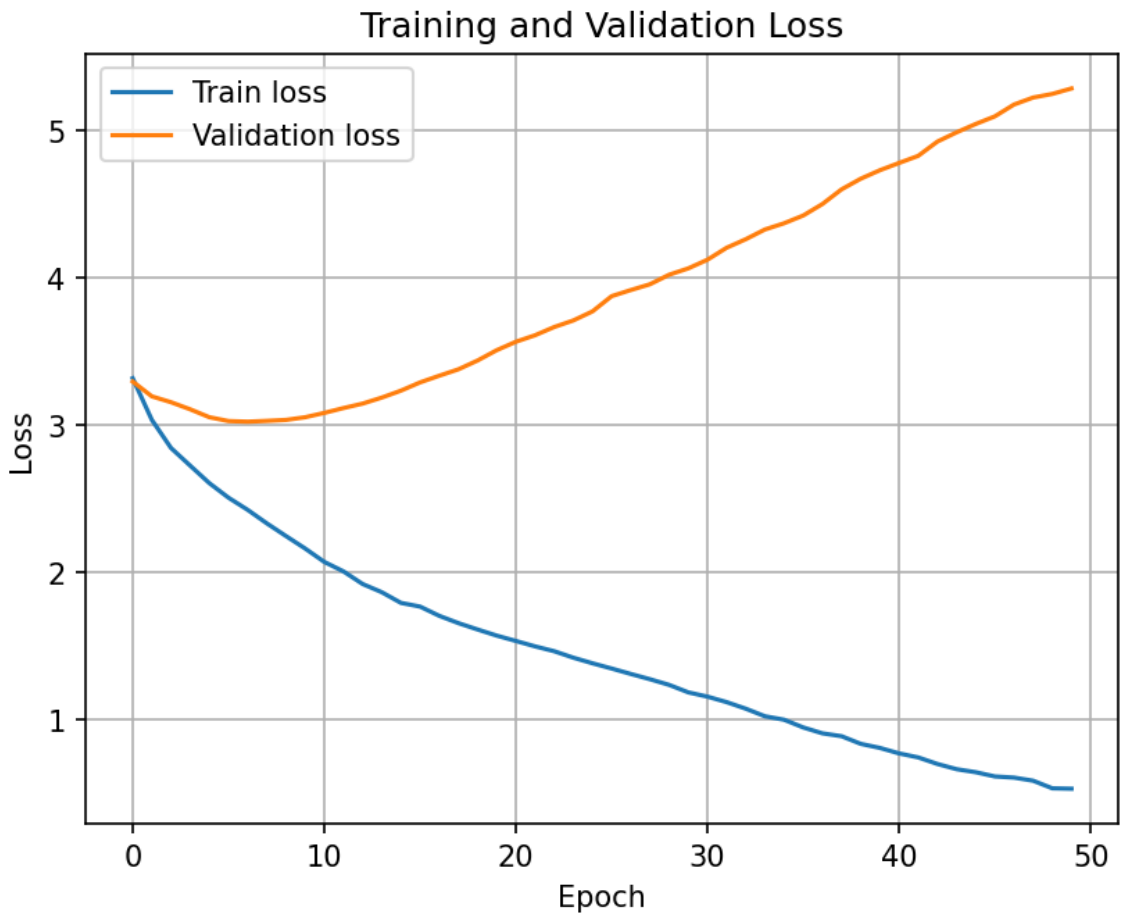
```
        top_k=top_k,
        num_samples=args.num_samples,
    )

    print("\n================= GENERATED TEXT ================\n")
    for i, text in enumerate(outputs, start=1):
        print(f"----- Sample {i} -----")
        print(text)
        print()


if __name__ == "__main__":
    main()
```

## 6. Experiments and Results

Below is the training vs validation loss curve:




### Generated Text Example:

Prompt: amar
Generated Output:
amar baladesh, ba boumi asha...

### Evaluation Matrics and Performance Analysis:

Cross-Entropy Loss was used as the primary evaluation metric in this project. It measures how well the model's predicted probability distribution matches the actual target distribution. A lower cross-entropy value indicates that the model's predictions are closer to the true distribution.

A secondary evaluation measure is Perplexity (PPL), computed as:
PPL = exp(cross_entropy_loss)

Perplexity represents how 'confused' the model is while predicting the next token. Lower perplexity means the model is more confident and accurate.

In our results, training loss steadily decreased, showing that the model learned the patterns present in the training data. However, validation loss increased over time, indicating overfitting due to the small dataset size. This shows that although the model memorized training patterns well, it struggled with generalization.

## 7. Conclusion

This report serves as both a technical guide and a beginner-friendly mini-book. A complete Transformer model was implemented from scratch using PyTorch, with detailed explanations.

## 8. References

1. Vaswani et al. (2017). Attention Is All You Need.
2. PyTorch Documentation.
3. Transformer research blogs & educational resources.