# Comparative Analysis of Music Transformer and Attention in Music Generation

Kane Wang (yw4623)  Mihir Agarwal (ma4874)

December 16, 2025

## 1 Introduction

The generation of music requires a model capable of maintaining long-term coherence and structure. While Transformer models have shown great success in sequence modeling, the standard "vanilla" self-attention mechanism utilizes absolute positional embeddings and requires $O(L^2)$ memory for a sequence of length $L$. This quadratic space complexity becomes a bottleneck for long musical sequences.

The primary goal of this project was to implement and evaluate the Music Transformer [1], which utilizes a relative attention mechanism to reduce space complexity, theoretically allowing for longer context windows and better structure preservation compared to vanilla attention.

## 2 Literature Review

We focused on two foundational papers that established the use of relative positioning in Transformers:

### 2.1 Self-Attention with Relative Position Representations [2]

Shaw et al. introduced the concept that the interaction between two tokens should depend on the distance between them (relative position) rather than their absolute positions in the sequence. They modeled the input as a labeled, directed graph where edges capture the relative position information.

- **Clipping Distance ($k$):** The authors hypothesized that precise relative position information is not useful beyond a certain distance. Therefore, they clipped the relative distance at a maximum value $k$, meaning the model learns unique representations for relative positions within $[-k, k]$ and treats all distances beyond that as equal.
- **Memory Limitation:** While effective for machine translation, the implementation by Shaw et al. requires creating an intermediate tensor of shape $(L, L, D)$, leading to a memory complexity of $O(L^2D)$, where $D$ is the hidden dimension. This additional factor of $D$ makes it computationally prohibitive for the very long sequences required in music generation.

### 2.2 Music Transformer [1]

Huang et al. addressed the memory bottleneck identified in Shaw et al.'s work. They proposed a memory-efficient "skewing" algorithm. Instead of creating the massive intermediate tensor for relative embeddings, this method computes relative attention logits directly. This optimization reduces the space complexity back to $O(L^2)$, enabling the processing of much longer sequences—a critical requirement for capturing musical structure over minutes of audio.

# 3 Methodology & Implementation

We implemented the architecture from scratch to compare the standard attention mechanism against the memory-efficient relative attention proposed by Huang et al.

## 3.1 Vanilla Attention (Baseline)

We trained a standard Transformer baseline using absolute positional encodings. This serves as the control to measure the benefits of the relative attention mechanism.

## 3.2 Music Transformer (Relative Attention)

We implemented the memory-efficient relative attention mechanism.

- **Mechanism:** We utilized the "skewing" procedure to transform the absolute position attention logits into relative position logits. This avoids the $O(L^2D)$ memory cost inherent in the naive implementation from Shaw et al. [2].
- **Benefit:** This allows the model to learn invariance to translation (a musical motif should have the same meaning regardless of where it appears in time) while maintaining a manageable memory footprint.

The specific implementation details and code for the Relative Global Attention mechanism can be found in **Appendix A**.

# 4 Experiments

Both the Vanilla Transformer and the Music Transformer were trained on the same dataset of musical sequences to ensure a fair comparison.

- **Training Environment:** GPU (A100)
- **Metrics:** Training Loss, Validation Loss, and perceptual quality of generated samples.

# 5 Results and Observations

Contrary to the theoretical superiority of the Music Transformer for long sequences, our experimental results yielded the following observations:

**Performance:** The Vanilla Attention model actually performed better in our tests, producing lower loss and subjectively more coherent musical phrases.

**Convergence:** The Vanilla model demonstrated more stable convergence during the training steps utilized.

## 6 Discussion and Future Scope

We analyzed why the Vanilla Attention outperformed the Music Transformer in this specific iteration.

### Window Size Limitations

The primary benefit of the Music Transformer is its space efficiency, which allows for significantly larger window sizes (context length). In our experiments, the window size may not have been large enough for the relative attention mechanism to demonstrate its advantage. At shorter sequence lengths, the overhead and complexity of relative attention might outweigh its benefits compared to the straightforward absolute positioning of Vanilla attention.

### Future Work

To fully realize the benefits of the Music Transformer paper, future work will focus on increasing the window size. By training on much longer sequences, the memory bottlenecks of Vanilla Attention will become apparent, likely allowing the Music Transformer's space-efficient relative attention to achieve superior performance in capturing long-term structure.

## References

[1] Huang, C.-Z. A., et al. (2018). *Music Transformer: Generating Music with Long-Term Structure*. arXiv:1809.04281

[2] Shaw, P., Uszkoreit, J., & Vaswani, A. (2018). *Self-Attention with Relative Position Representations*. Proceedings of NAACL-HLT 2018, pp. 464–468. https://aclanthology.org/N18-2074/

College students get the Pro plan of Google Gemini and Colab free. Terms Apply. Learn more at gemini.google/students and colab.research.google.com/signup Learn more

MusicTransformer.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Connect A100 High-RAM

The Notebook consists of experiment on relational positional embedding using the J.S. Bach Chorales Dataset

```
import os
from pathlib import Path
import pickle
import random
import math
import logging
from typing import List, Dict, Optional, Tuple
from collections import Counter

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.optim import AdamW
from torch.optim.lr_scheduler import LambdaLR  # or other schedulers

from transformers import BertConfig, BertForMaskedLM, get_linear_schedule_with_warmup, set_seed
from tqdm.auto import tqdm

# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
Using device: cuda
```

```
# Load the JSB Chorales dataset

def load_jsb_chorales(pkl_path):
    """
    Load the pickled JSB Chorales dataset.
    Converts any scalar values in sequences to single-element arrays.
    """
    with open(pkl_path, 'rb') as f:
        data = pickle.load(f, encoding='latin1')

    for split in ('train', 'valid', 'test'):
        if split not in data:
            continue

        for p_idx in range(len(data[split])):
            piece = data[split][p_idx]
            new_piece = []

            for timestep in piece:
                # Check if it's a scalar
                if np.isscalar(timestep) or isinstance(timestep, (np.integer, np.floating, int, float)):
                    new_piece.append(np.array([timestep]))
                else:
                    # It's already a tuple/list/array
                    new_piece. append(np.array(timestep))

            data[split][p_idx] = new_piece

    return data

# Upload your . pkl file in Colab
from google.colab import files
uploaded = files.upload()  # Upload your . pkl file here
pkl_filename = list(uploaded.keys())[0]

data = load_jsb_chorales(pkl_filename)
print(f"Data keys: {data.keys()}")
print(f"Train samples: {len(data['train'])}")
print(f"Valid samples: {len(data['valid'])}")
print(f"Test samples: {len(data['test'])}")
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving jsb-chorales-16th.pkl to jsb-chorales-16th (1).pkl
Data keys: dict_keys(['test', 'train', 'valid'])
Train samples: 229
Valid samples: 76
Test samples: 77
```

```
def tuple_list_to_array(tuple_list):
    """
    Convert nested structure to (T,4) array.
    Handles scalars, 1D arrays, and tuples.
    """
    flat = []
    for sub in tuple_list:
        # Handle if sub is a scalar (wrapped or unwrapped)
        if np.isscalar(sub) or (isinstance(sub, np.ndarray) and sub.ndim == 0):
            flat.append(int(sub))
        # Handle if sub is already a 1D array or tuple with multiple elements
```

```python
        elif isinstance(sub, (list, tuple, np.ndarray)):
            # If it's a 1D array with one element, unwrap it
            if isinstance(sub, np.ndarray) and sub.shape == (1,):
                flat.append(int(sub[0]))
            else:
                # It's a proper tuple/list/array of values
                for t in sub:
                    flat. append(int(t))
        else:
            flat.append(int(sub))

    arr = np.asarray(flat, dtype=int)

    # Reshape into (T,4) when divisible by 4
    if arr. ndim == 1 and arr.size % 4 == 0:
        arr = arr. reshape(-1, 4)

    return arr

def serialize_timegrid(arr_Tx4, add_special_tokens=True, start_token=0, end_token=1):
    """
    Serialize (T,4) -> 1D sequence with optional special tokens.
    If add_special_tokens=True:
      [START_TOKEN, S1, A1, T1, B1, S2, A2, T2, B2, .. ., END_TOKEN]
    """
    arr = tuple_list_to_array(arr_Tx4)
    serialized = arr.reshape(-1).astype(int)

    if add_special_tokens:
        # Add start token at beginning and end token at end
        serialized = np.concatenate([[start_token], serialized, [end_token]])

    return serialized

def serialize_all_pieces(pieces_list, add_special_tokens=True, start_token=0, end_token=1):
    """
    Serialize all pieces in the dataset.
    Returns: list of serialized 1D numpy arrays (one per piece).
    Each piece gets START and END tokens if add_special_tokens=True.
    """
    serialized = []
    for piece in pieces_list:
        arr = serialize_timegrid(piece, add_special_tokens=add_special_tokens,
                                 start_token=start_token, end_token=end_token)
        serialized.append(arr)
    return serialized
```

[ ]

```python
# Vocabulary

def build_vocab_from_serialized(serialized_seqs, specials=None):
    """
    Returns:
      token2id : dict mapping special token strings AND numeric pitch integers -> integer ids
                 (So keys are mixed: strings for specials, ints for pitches)
      id2token : list mapping id -> token (special strings first, then numeric pitch integers)
      pitch_to_id: dict mapping numeric pitch (int) -> id
    Notes:
      - <PAD> will be reserved if included in specials (default set includes it).
      - This keeps pitch tokens as numeric values (not string "P_60"), matching your data structure.
    """
    if specials is None:
        specials = ["<PAD>", "<MASK>", "<CLS>", "<SEP>", "<UNK>"]
    counter = Counter()
    for s in serialized_seqs:
        counter.update(list(s))
    # convert keys to ints
    pitches = sorted({int(x) for x in counter.keys()})
    id2token = []
    token2id = {}
    # Reserve specials first in the order given (keys are strings)
    for tok in specials:
        token2id[tok] = len(id2token)
        id2token.append(tok)
    # Map numeric pitch values directly to token ids (keys are ints)
    for p in pitches:
        token2id[int(p)] = len(id2token)   # allow integer key
        id2token.append(int(p))            # id2token stores the numeric pitch
    # Helper mapping pitch integer -> token id
    pitch_to_id = {p: token2id[p] for p in pitches}
    return token2id, id2token, pitch_to_id

# Convert serialized integer pitch sequence to token ids using pitch_to_id mapping
def serialized_pitch_seq_to_ids(serialized_seq, pitch_to_id, unk_token="<UNK>", token2id=None):
    """
    serialized_seq: 1D iterable of pitch numbers (numpy ints allowed)
    pitch_to_id: mapping int pitch -> id (from build_vocab_from_serialized)
    token2id: required to get unk index (token2id["<UNK>"])
    """
    if token2id is None:
        raise ValueError("Provide token2id (to get unk index).")
    unk_idx = token2id[unk_token]
    # map each element to int then to id; unknown pitches -> unk_idx
    ids = np.array([pitch_to_id.get(int(p), unk_idx) for p in serialized_seq], dtype=np.int64)
    return ids
```

[ ]

```python
# Windows
def windowize(ids_1d, seq_len=1024, pad_id=0, stride=None):
    """
    ids_1d: 1D np array of token ids
```

```
    seq_len: target length in tokens (including special tokens if used)
    stride: if None -> non-overlapping windows, else overlapping with given stride
    returns list of windows (np arrays length seq_len)
    """
    if stride is None:
        stride = seq_len
    out = []
    n = len(ids_1d)
    i = 0
    while i < n:
        w = ids_1d[i:i+seq_len]
        if len(w) < seq_len:
            pad = np.full(seq_len - len(w), pad_id, dtype=np.int64)
            w = np.concatenate([w, pad])
        out.append(w)
        i += stride
    if not out:
        out.append(np.full(seq_len, pad_id, dtype=np.int64))
    return out
```

[ ]

```
def create_masked_input_from_ids(input_ids_np,
                                 token2id,
                                 mlm_prob=0.15,
                                 mask_token="<MASK>",
                                 unk_token="<UNK>",
                                 pad_token="<PAD>",
                                 mode="token"):
    """
    input_ids_np: 1D numpy array of token ids for a window (no special handling)
    mode: "token" => mask tokens independently.
          "time"  => mask at time-step granularity; since each time-step has 4 tokens in consecutive groups,
                     treat chunks of 4 tokens as a unit and select ~ mlm_prob fraction of those units to mask.
    returns:
      input_ids_masked (np.array): masked input ids
      labels (np.array): original ids for masked positions, -100 elsewhere (PyTorch ignore index)
    """
    input_ids = input_ids_np.copy()
    labels = np.full(input_ids.shape, -100, dtype=np.int64)
    vocab_size = len(token2id)
    pad_idx = token2id[pad_token]
    mask_idx = token2id[mask_token]
    unk_idx = token2id[unk_token]

    n = len(input_ids)
    # Maskable positions: exclude pad token
    maskable = (input_ids != pad_idx)
    if mode == "token":
        candidate_pos = np.where(maskable)[0]
        n_to_mask = max(1, int(round(len(candidate_pos) * mlm_prob)))
        mask_pos = np.random.choice(candidate_pos, size=n_to_mask, replace=False)
        for pos in mask_pos:
            orig = input_ids[pos]
            labels[pos] = orig
            r = random.random()
            if r < 0.8:
                input_ids[pos] = mask_idx
            elif r < 0.9:
                # replace with random token (not pad)
                rand = random.randrange(vocab_size)
                # avoid choosing PAD token as replacement to keep training signal reasonable
                if rand == pad_idx:
                    rand = (rand + 1) % vocab_size
                input_ids[pos] = rand
            else:
                # keep original
                pass
    elif mode == "time":
        # assume groups of 4 tokens correspond to one time step:
        time_steps = n // 4
        # maskable time steps are those where at least one token in group != PAD
        maskable_ts = []
        for t in range(time_steps):
            group = input_ids[4*t:4*t+4]
            if np.any(group != pad_idx):
                maskable_ts.append(t)
        n_to_mask = max(1, int(round(len(maskable_ts) * mlm_prob)))
        ts_to_mask = np.random.choice(maskable_ts, size=n_to_mask, replace=False)
        for t in ts_to_mask:
            for pos in range(4*t, 4*t+4):
                if input_ids[pos] == pad_idx:
                    continue
                orig = input_ids[pos]
                labels[pos] = orig
                r = random.random()
                if r < 0.8:
                    input_ids[pos] = mask_idx
                elif r < 0.9:
                    rand = random.randrange(vocab_size)
                    if rand == pad_idx:
                        rand = (rand + 1) % vocab_size
                    input_ids[pos] = rand
                else:
                    pass
    else:
        raise ValueError("Unknown mode")
    return input_ids, labels
```

[ ]

```
# Dataset
class MusicMLMDataset(Dataset):
    def __init__(self, windows_ids_list, token2id, seq_len=1024, mlm_prob=0.15,
```

```python
    def __init__(self, windows_ids_list, token2id, seq_len=1024, mlm_prob=0.15,
                 mode="token", do_add_cls=False, cls_token="<CLS>", sep_token="<SEP>"):
        """
        windows_ids_list: list of np arrays each length seq_len (token ids)
        token2id: mapping (contains both special-string keys and integer pitch keys)
        mode: "token" or "time"
        do_add_cls: whether to prefix with CLS (then last token possibly dropped to preserve seq_len)
        """
        self.windows = [np.array(w, dtype=np.int64) for w in windows_ids_list]
        self.token2id = token2id
        self.seq_len = seq_len
        self.mlm_prob = mlm_prob
        self.mode = mode
        self.pad_id = token2id["<PAD>"]
        self.cls_token = cls_token
        self.sep_token = sep_token
        self.do_add_cls = do_add_cls

    def __len__(self):
        return len(self.windows)

    def __getitem__(self, idx):
        ids = self.windows[idx].copy()
        # Optionally add CLS at position 0 (shift right and drop last token so length unchanged)
        if self.do_add_cls:
            cls_id = self.token2id[self.cls_token]
            ids = np.concatenate([[cls_id], ids[:-1]])
        input_ids_masked, labels = create_masked_input_from_ids(
            ids,
            token2id=self.token2id,
            mlm_prob=self.mlm_prob,
            mode=self.mode
        )
        attention_mask = (ids != self.pad_id).astype(np.int64)
        # Convert to torch tensors
        return {
            "input_ids": torch.from_numpy(input_ids_masked).long(),
            "labels": torch.from_numpy(labels).long(),
            "attention_mask": torch.from_numpy(attention_mask).long()
        }
```

[ ]

```python
def preprocess_dataset_splits(data: Dict[str, List],
                              seq_len: int=1024,
                              stride: int=None,
                              build_vocab_from: str="train",   # "train" or "all"
                              mask_mode: str="time",
                              do_add_cls: bool=False) -> Tuple[Dict, List, Dict, Dict[str, List[np.ndarray]]]:
    """
    data: dictionary-like object that contains splits. Expected keys (case-insensitive): 'train', 'test', 'valid' or 'validation'.
          Each value should be a list of pieces; each piece is shape (T,4) or list of T 4-tuples.
    seq_len, stride: windowing params (stride in tokens; pass stride=None for non-overlapping windows)
    build_vocab_from: "train" (recommended) or "all" - where to collect pitch types for vocab
    Returns:
      token2id, id2token, pitch_to_id, processed_windows_per_split dict with keys 'train','validation','test'
    """
    # Normalize keys and pick available splits
    lower_map = {k.lower(): k for k in data.keys()}
    # support both 'valid' and 'validation' synonyms
    split_keys = {}
    for name in ("train", "valid", "test"):
        if name in lower_map:
            split_keys[name] = lower_map[name]

    # Helper to extract pieces list or empty
    def get_split(name):
        key = split_keys.get(name)
        return data[key] if (key is not None) else []

    # 1) Serialize pieces for splits we have (but we may build vocab from train only)
    serialized = {}
    for s in ['train', 'valid', 'test']:
        pieces = get_split(s)
        if pieces:
            serialized[s] = serialize_all_pieces(pieces)
        else:
            serialized[s] = []

    # 2) Build vocab from requested source
    if build_vocab_from == "train":
        vocab_source = serialized['train']
    elif build_vocab_from == "all":
        vocab_source = serialized['train'] + serialized['valid'] + serialized['test']
    else:
        raise ValueError("build_vocab_from must be 'train' or 'all'")

    token2id, id2token, pitch_to_id = build_vocab_from_serialized(vocab_source)
    pad_id = token2id["<PAD>"]

    # 3) Convert serialized pitch sequences to token ids and windowize per split
    processed = {}
    for s in ['train', 'valid', 'test']:
        windows = []
        for seq in serialized[s]:
            ids = serialized_pitch_seq_to_ids(seq, pitch_to_id, unk_token="<UNK>", token2id=token2id)
            ws = windowize(ids, seq_len=seq_len, pad_id=pad_id, stride=stride)
            windows.extend(ws)
        processed[s] = windows

    return token2id, id2token, pitch_to_id, processed
```

[ ]

```python
seq_len = 16
```

```
stride = 1  # overlap example

token2id, id2token, pitch_to_id, processed = preprocess_dataset_splits(
    data,
    seq_len=seq_len,
    stride=stride,
    build_vocab_from="train",
    mask_mode="time",
    do_add_cls=False
)

token2id, id2token, pitch_to_id, processed = preprocess_dataset_splits(
    data,
    seq_len=seq_len,
    stride=stride,
    build_vocab_from="train",
    mask_mode="time",
    do_add_cls=False
)

print("Vocab size (id2token):", len(id2token))
print("Processed windows counts:")
for k in ('train', 'valid', 'test'):
    print(f"  {k}: {len(processed[k])}")

# Create dataset and dataloader for the train split
train_windows = processed['train']
dataset = MusicMLMDataset(train_windows, token2id, seq_len=seq_len, mlm_prob=0.15, mode="time", do_add_cls=False)
loader = DataLoader(dataset, batch_size=4, shuffle=True)
batch = next(iter(loader))
print(batch["input_ids"].shape, batch["labels"].shape, batch["attention_mask"].shape)
# Use batch["input_ids"] as model input, compute loss with CrossEntropyLoss(ignore_index=-100) against batch["labels"].
```

```
Vocab size (id2token): 53
Processed windows counts:
  train: 220959
  valid: 73195
  test: 75470
torch.Size([4, 16]) torch.Size([4, 16]) torch.Size([4, 16])
```

```
# Create datasets
train_dataset = MusicMLMDataset(
    processed['train'], token2id, seq_len=seq_len,
    mlm_prob=0.15, mode="time", do_add_cls=False
)
valid_dataset = MusicMLMDataset(
    processed['valid'], token2id, seq_len=seq_len,
    mlm_prob=0.15, mode="time", do_add_cls=False
)
test_dataset = MusicMLMDataset(
    processed['test'], token2id, seq_len=seq_len,
    mlm_prob=0.15, mode="time", do_add_cls=False
)

# Create dataloaders
batch_size = 8
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

print(f"Train batches: {len(train_loader)}")
print(f"Valid batches: {len(valid_loader)}")
print(f"Test batches: {len(test_loader)}")
```

```
Train batches: 27620
Valid batches: 9150
Test batches: 9434
```

```
# Relative Positional Representation with Skewing Algorithm (O(LD) complexity)
import math
import torch
import torch.nn as nn
import torch. nn.functional as F
from transformers import BertForMaskedLM, BertConfig


class RelativePositionBias(nn.Module):
    """
    Relative position embeddings for the skewing algorithm from Music Transformer.
    Creates learnable embeddings for relative distances.
    """
    def __init__(self, max_relative_position, d_model):
        super().__init__()
        self.max_relative_position = max_relative_position
        self. d_model = d_model

        # Embeddings for relative positions:  [-max_rel, ..., -1, 0, 1, ..., max_rel]
        vocab_size = 2 * max_relative_position + 1
        self. embeddings = nn.Embedding(vocab_size, d_model)

    def forward(self, length):
        """
        Generate relative position embeddings for sequence of given length.
        Returns: (2*length - 1, d_model)
        """
        # We need embeddings for relative distances from -(length-1) to +(length-1)
        rel_positions = torch.arange(
            -(length - 1),
            length,
            device=self.embeddings. weight.device
        )
```

```python
        # Clamp to max relative position and shift to positive indices
        rel_positions_clipped = torch.clamp(
            rel_positions,
            -self. max_relative_position,
            self.max_relative_position
        ) + self.max_relative_position

        # Get embeddings:  (2*length - 1, d_model)
        rel_embeddings = self.embeddings(rel_positions_clipped)

        return rel_embeddings


def _skewing(tensor):
    """
    Skewing algorithm from Music Transformer paper.
    Converts relative position tensor from shape (..., length, 2*length - 1) to (..., length, length).
    This efficiently maps relative position logits to absolute positions.

    Input: (batch, heads, length, 2*length - 1)
    Output: (batch, heads, length, length)
    """
    batch_size, num_heads, length, _ = tensor. shape

    # Pad with zeros on the left:  add one column
    # (batch, heads, length, 2*length - 1) -> (batch, heads, length, 2*length)
    padded = F.pad(tensor, (1, 0))

    # Reshape: (batch, heads, length, 2*length) -> (batch, heads, 2*length, length)
    reshaped = padded.reshape(batch_size, num_heads, 2 * length, length)

    # Slice: take rows from 'length' onwards
    # (batch, heads, 2*length, length) -> (batch, heads, length, length)
    skewed = reshaped[: , :, length:, :]

    return skewed


def _relative_attention_inner(query, key, rel_embeddings):
    """
    Compute relative position attention using skewing algorithm.
    This is the memory-efficient O(LD) implementation from Music Transformer.

    Args:
        query:  (batch, heads, length, d_k)
        key: (batch, heads, length, d_k)
        rel_embeddings: (2*length - 1, d_k) relative position embeddings

    Returns:
        logits: (batch, heads, length, length) attention logits with relative positions
    """
    batch_size, num_heads, length, d_k = query.shape

    # Absolute position logits: Q * K^T
    logits_absolute = torch.matmul(query, key.transpose(-2, -1))

    # Relative position logits using skewing
    # Step 1: Compute Q * E_r^T where E_r are relative position embeddings
    # Reshape query for batch matmul:  (batch * heads, length, d_k)
    query_flat = query.reshape(batch_size * num_heads, length, d_k)

    # Compute Q * E_r^T: (batch * heads, length, d_k) x (d_k, 2*length-1)
    # -> (batch * heads, length, 2*length-1)
    rel_logits = torch. matmul(query_flat, rel_embeddings.transpose(0, 1))

    # Reshape back: (batch, heads, length, 2*length - 1)
    rel_logits = rel_logits. reshape(batch_size, num_heads, length, 2 * length - 1)

    # Step 2: Apply skewing to convert (batch, heads, length, 2*length-1)
    # -> (batch, heads, length, length)
    rel_logits = _skewing(rel_logits)

    # Combine absolute and relative logits
    return logits_absolute + rel_logits


def create_relative_forward_function(rel_pos_embeddings, num_attention_heads, attention_head_size, all_head_size):
    """
    Create a forward function for relative attention.
    """
    def forward_fn(
        self,
        hidden_states,
        attention_mask=None,
        head_mask=None,
        encoder_hidden_states=None,
        encoder_attention_mask=None,
        past_key_value=None,
        output_attentions=False,
        **kwargs
    ):
        batch_size, seq_len, hidden_size = hidden_states. size()

        # Q, K, V projections
        mixed_query_layer = self.query(hidden_states)
        mixed_key_layer = self.key(hidden_states)
        mixed_value_layer = self.value(hidden_states)

        # Reshape and transpose for multi-head attention
        def transpose_for_scores(x):
            new_x_shape = x.size()[:-1] + (num_attention_heads, attention_head_size)
```

```python
            x = x.view(new_x_shape)
            return x. permute(0, 2, 1, 3)

        query_layer = transpose_for_scores(mixed_query_layer)
        key_layer = transpose_for_scores(mixed_key_layer)
        value_layer = transpose_for_scores(mixed_value_layer)

        # Get relative position embeddings for this sequence length
        rel_embeddings = rel_pos_embeddings(seq_len)

        # Compute attention scores with relative positions using skewing (O(LD) complexity)
        attention_scores = _relative_attention_inner(query_layer, key_layer, rel_embeddings)

        # Scale
        attention_scores = attention_scores / math.sqrt(attention_head_size)

        # Apply attention mask
        if attention_mask is not None:
            attention_scores = attention_scores + attention_mask

        # Normalize
        attention_probs = nn.functional.softmax(attention_scores, dim=-1)
        attention_probs = self.dropout(attention_probs)

        # Apply head mask if provided
        if head_mask is not None:
            attention_probs = attention_probs * head_mask

        # Compute context
        context_layer = torch.matmul(attention_probs, value_layer)
        context_layer = context_layer.permute(0, 2, 1, 3).contiguous()
        new_context_layer_shape = context_layer.size()[:-2] + (all_head_size,)
        context_layer = context_layer. view(new_context_layer_shape)

        outputs = (context_layer, attention_probs) if output_attentions else (context_layer,)

        return outputs

    return forward_fn


def create_bert_with_relative_positions(config):
    """
    Create a BertForMaskedLM model with relative positional encoding using skewing algorithm.
    This is the O(LD) memory-efficient implementation from Music Transformer paper.
    Only modifies the attention mechanism - everything else identical to baseline.
    """
    # Create a new config with relative positions
    new_config = BertConfig(
        vocab_size=config. vocab_size,
        hidden_size=config.hidden_size,
        num_hidden_layers=config.num_hidden_layers,
        num_attention_heads=config.num_attention_heads,
        intermediate_size=config. intermediate_size,
        max_position_embeddings=config.max_position_embeddings,
        pad_token_id=config. pad_token_id,
        mask_token_id=config. mask_token_id,
        position_embedding_type="relative_key"
    )

    # Create BERT model with new config
    model = BertForMaskedLM(new_config)

    # Calculate dimensions
    d_k = new_config.hidden_size // new_config.num_attention_heads
    all_head_size = new_config.num_attention_heads * d_k

    # Replace each attention layer's forward method
    for layer_idx, layer in enumerate(model. bert.encoder.layer):
        self_attn = layer.attention.self

        # Create relative position embeddings for this layer
        rel_pos_emb = RelativePositionBias(
            max_relative_position=new_config.max_position_embeddings,
            d_model=d_k
        )

        # Store it as a module attribute so it's registered properly
        setattr(self_attn, 'relative_position_embeddings', rel_pos_emb)

        # Create and bind the new forward function
        forward_fn = create_relative_forward_function(
            rel_pos_emb,
            new_config.num_attention_heads,
            d_k,
            all_head_size
        )

        # Bind as instance method
        import types
        self_attn.forward = types. MethodType(forward_fn, self_attn)

    return model


def create_baseline_bert(config):
    """
    Create baseline BERT model (standard absolute positions).
    """
    # Create a new config with absolute positions
    baseline_config = BertConfig(
        vocab_size=config.vocab_size,
```

```python
        vocab_size=config.vocab_size,
        hidden_size=config.hidden_size,
        num_hidden_layers=config.num_hidden_layers,
        num_attention_heads=config.num_attention_heads,
        intermediate_size=config.intermediate_size,
        max_position_embeddings=config.max_position_embeddings,
        pad_token_id=config.pad_token_id,
        mask_token_id=config.mask_token_id,
        position_embedding_type="absolute"
    )

    return BertForMaskedLM(baseline_config)
```

[ ]

```python
# Same config for both models
vocab_size = len(id2token)
config = BertConfig(
    vocab_size=vocab_size,
    hidden_size=128,
    num_hidden_layers=2,
    num_attention_heads=4,
    intermediate_size=512,
    max_position_embeddings=128,
    pad_token_id=token2id["<PAD>"],
    mask_token_id=token2id["<MASK>"],
)

# Create baseline model (standard BERT from transformers)
print("Creating BASELINE model...")
model_baseline = create_baseline_bert(config)
model_baseline = model_baseline.to(device)
baseline_params = sum(p.numel() for p in model_baseline.parameters())
print(f"Baseline parameters: {baseline_params:,}")

# Create relative position model (same architecture, only attention changed)
print("\nCreating RELATIVE POSITION model...")
model_relative = create_bert_with_relative_positions(config)
model_relative = model_relative.to(device)
relative_params = sum(p.numel() for p in model_relative.parameters())
print(f"Relative parameters: {relative_params:,}")

print(f"\nParameter difference: {relative_params - baseline_params:,}")
```

```
Creating BASELINE model...
Baseline parameters: 437,045

Creating RELATIVE POSITION model...
Relative parameters: 469,813

Parameter difference: 32,768
```

[ ]

```python
# Minimal training hyperparameters
num_epochs = 10              # Reduced from 50
learning_rate = 5e-4
warmup_steps = 100           # Reduced from 500
accumulation_steps = 4       # Simulate larger batch via accumulation

total_steps = len(train_loader) * num_epochs // accumulation_steps
```

[ ]

```python
def train_epoch(model, dataloader, optimizer, scheduler, device, accumulation_steps=4):
    model.train()
    total_loss = 0
    optimizer.zero_grad()

    progress_bar = tqdm(dataloader, desc="Training")

    for i, batch in enumerate(progress_bar):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["labels"].to(device)

        # Forward pass
        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            labels=labels
        )
        loss = outputs.loss / accumulation_steps  # Scale loss

        # Backward pass
        loss.backward()

        # Update weights every accumulation_steps
        if (i + 1) % accumulation_steps == 0:
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

        total_loss += loss. item() * accumulation_steps
        progress_bar.set_postfix({"loss": loss.item() * accumulation_steps})

        # Clear cache periodically
        if i % 50 == 0:
            torch.cuda.empty_cache()

    return total_loss / len(dataloader)

def evaluate(model, dataloader, device):
    model.eval()
    total_loss = 0
```

```python
        with torch.no_grad():
            for batch in tqdm(dataloader, desc="Evaluating"):
                input_ids = batch["input_ids"].to(device)
                attention_mask = batch["attention_mask"].to(device)
                labels = batch["labels"].to(device)

                outputs = model(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    labels=labels
                )
                total_loss += outputs.loss.item()

        torch.cuda.empty_cache()
        return total_loss / len(dataloader)
```

[ ]

```python
    # Train relative position model
    print("\n" + "="*60)
    print("TRAINING RELATIVE POSITION MODEL")
    print("="*60)

    optimizer_relative = AdamW(model_relative.parameters(), lr=learning_rate, weight_decay=0.01)
    scheduler_relative = get_linear_schedule_with_warmup(
        optimizer_relative, num_warmup_steps=warmup_steps, num_training_steps=total_steps
    )

    best_valid_loss = float('inf')
    for epoch in range(num_epochs):
        print(f"\nEpoch {epoch + 1}/{num_epochs}")
        train_loss = train_epoch(model_relative, train_loader, optimizer_relative, scheduler_relative, device, accumulation_steps)
        valid_loss = evaluate(model_relative, valid_loader, device)
        print(f"Train Loss: {train_loss:.4f}, Valid Loss: {valid_loss:.4f}")

        if valid_loss < best_valid_loss:
            best_valid_loss = valid_loss
            torch.save(model_relative.state_dict(), "relative_bert.pt")

    model_relative.load_state_dict(torch.load("relative_bert.pt"))
    relative_test_loss = evaluate(model_relative, test_loader, device)
    print(f"\nRelative Position Test Loss: {relative_test_loss:.4f}")


    # Train baseline
    print("\n" + "="*60)
    print("TRAINING BASELINE MODEL")
    print("="*60)

    optimizer_baseline = AdamW(model_baseline.parameters(), lr=learning_rate, weight_decay=0.01)
    scheduler_baseline = get_linear_schedule_with_warmup(
        optimizer_baseline, num_warmup_steps=warmup_steps, num_training_steps=total_steps
    )

    best_valid_loss = float('inf')
    for epoch in range(num_epochs):
        print(f"\nEpoch {epoch + 1}/{num_epochs}")
        train_loss = train_epoch(model_baseline, train_loader, optimizer_baseline, scheduler_baseline, device, accumulation_steps)
        valid_loss = evaluate(model_baseline, valid_loader, device)
        print(f"Train Loss: {train_loss:.4f}, Valid Loss: {valid_loss:.4f}")

        if valid_loss < best_valid_loss:
            best_valid_loss = valid_loss
            torch.save(model_baseline.state_dict(), "baseline_bert.pt")

    model_baseline.load_state_dict(torch.load("baseline_bert.pt"))
    baseline_test_loss = evaluate(model_baseline, test_loader, device)
    print(f"\nBaseline Test Loss: {baseline_test_loss:.4f}")



    # Final comparison
    print("\n" + "="*60)
    print("FINAL COMPARISON")
    print("="*60)
    print(f"Baseline Test Loss:           {baseline_test_loss:.4f}")
    print(f"Relative Position Test Loss:  {relative_test_loss:.4f}")
    print(f"Improvement:                 {baseline_test_loss - relative_test_loss:.4f}")
```

```
Epoch 4/10
Training: 100% 27620/27620 [03:39<00:00, 117.77it/s, loss=0.323]
Evaluating: 100% 9150/9150 [00:26<00:00, 348.68it/s]
Train Loss: 0.5082, Valid Loss: 0.4539

Epoch 5/10
Training: 100% 27620/27620 [03:40<00:00, 126.25it/s, loss=0.193]
Evaluating: 100% 9150/9150 [00:26<00:00, 351.67it/s]
Train Loss: 0.4801, Valid Loss: 0.4324

Epoch 6/10
Training: 100% 27620/27620 [03:40<00:00, 127.16it/s, loss=0.814]
Evaluating: 100% 9150/9150 [00:26<00:00, 348.80it/s]
Train Loss: 0.4611, Valid Loss: 0.4167

Epoch 7/10
Training: 100% 27620/27620 [03:41<00:00, 126.52it/s, loss=0.253]
Evaluating: 100% 9150/9150 [00:26<00:00, 338.17it/s]
Train Loss: 0.4458, Valid Loss: 0.4033
```

```
Train Loss: 0.4458, Valid Loss: 0.4022

Epoch 8/10
Training: 100% 27620/27620 [03:40<00:00, 126.75it/s, loss=0.519]
Evaluating: 100% 9150/9150 [00:26<00:00, 344.44it/s]
Train Loss: 0.4310, Valid Loss: 0.3936

Epoch 9/10
Training:  43% 11971/27620 [01:35<02:06, 123.94it/s, loss=0.348]
Evaluating:   0%|          | 0/9150 [00:00<?, ?it/s]
Train Loss: 0.4216, Valid Loss: 0.3838

Epoch 10/10
Training: 100% 27620/27620 [03:41<00:00, 128.36it/s, loss=0.29]
Evaluating: 100% 9150/9150 [00:26<00:00, 350.57it/s]
Train Loss: 0.4109, Valid Loss: 0.3777
Evaluating: 100% 9434/9434 [00:27<00:00, 348.01it/s]

Baseline Test Loss: 0.4014

============================================================
FINAL COMPARISON
============================================================
Baseline Test Loss:           0.4014
Relative Position Test Loss:  0.4865
Improvement:                 -0.0851
```