DECLARATION

I understand that this is an individual assessment and that collaboration is not permitted. I have not received any assistance with my work for this assessment. Where I have used the published work of others, I have indicated this with appropriate citation.

I have not and will not share any part of my work on this assessment, directly or indirectly, with any other student.

I have read and understood the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also read and understood the guide, and completed the 'Ready Steady Write' Tutorial on avoiding plagiarism, located at https://libguides.tcd.ie/academic-integrity/ready-steady-write.

I understand that by returning this declaration with my work, I am agreeing with the above statement.

Name: **AKSHAY ARORA(24338068)**

Date: **3rd January 2025**

1 Introduction

This document concerns the Final Assignment from the CS7CS4 Machine Learning 2024-25 module. The provided questions and respective details/answers are mentioned in the next section. The detailed code is mentioned in the Appendix (Section 4) and included in the zip file containing an executable Python script and the required dataset files.

2 Questions and related answers

PART 1

Q. (1) Repurpose the GPT text model provided with minimal changes so that it can generate melodies. The music dataset provided is the file 'inputMelodiesAugmented.txt'. Indicate the evaluation metrics you plan to use and explain how they are appropriate for this task. Your evaluation should include an appropriate baseline.

The given '*inputMelodiesAugmented.txt*' dataset was loaded in the provided GPT model and with minimal changes to the code, melodies were generated. Here, number of iterations(*max_iters*) were reduced to *1000* from *5000* due to limited available resources. *Table 1* shows the hyperparameters using in the GPT model training.

batch_size	block_size	max_iters	eval_interval	learning_rate	eval_iters	n_embd	n_head	n_layer	dropout
64	256	1000	500	3e-4	200	384	6	6	0.2

Table 1: Hyperparameters used in GPT Model

Once, the dataset is loaded, the unique characters are extracted i.e. the different notes in all the melodies and *R* representing the rest note.

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

```
Characters:['\n','A','B','C','D','E','F','G','R','a','c','d','f','g']
```

Post that, the GPT Model architecture is created and the number of parameters in the model recorded. These parameters are the result of combination of the chosen hyperparameters like embedding dimensions, no of layers/heads, etc. From, the given hyperparameters in the code, the total number of parameters turned out to be `10.74971 M`.

Then, the model is trained for *1000* iterations with loss being estimated at *step 0, 500 and 999.* At the final step, the losses came out to be `train loss 1.3361, val loss 1.3167`. The loss values suggest that that the model is relatively well trained, but not perfectly, and the validation loss being fairly close to the training loss suggests that the model is not overfitting and is generalizing well consistently on both training and unseen validation data. Post training completion, a melody of 500 tokens is generated and printed, sharing a subset below with the code for reference.

```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
gpt_melody = decode(m.generate(context, max_new_tokens=500)[0].tolist())
print(gpt_melody)
```

```
DCaRDCCCCDCFGddEFFEFADCDCDCDCACCAFAFFFADCCCCDCDECDCC
```

Now, before performing any evaluations, appropriate baseline model(s) is required for comparison. Here, three separate baseline models, having different strategies for generating melodies, are selected for thorough comparison:

- **Random Model** - In a random model, each token is selected at random from the vocabulary as it does not consider any patterns or structure in the data. Random melodies will be entirely disorganized and lack structure but can serve as a baseline for comparison. They will serve as a simple, non-intelligent benchmark for this GPT model for melodies.

```
def generate_random_melody(length, vocab):
    return ''.join(random.choices(vocab, k=length))
```

- **Frequency Based Model** - This model generates tokens based on their frequency in the training data as it creates melodies by picking the most common note observed in the dataset and repeating them. Tokens that occur more frequently in the dataset will have a higher probability of being chosen. The generated melodies from this baseline model might reflect common patterns, but they will lack long-range dependencies. This is a more straightforward approach to melody generation as by focusing on the most common notes, it creates melodies that reflect the common patterns in the training data.

```
def generate_freq_based_melody(length, freq_probs):
    vocab, probs = zip(*freq_probs.items())
    return ''.join(random.choices(vocab, probs, k=length))
```

- **N-Gram Model[3]** - The n-gram model is based on the idea of generating the next note based on the preceding n-notes. It uses a sliding window approach, where the model looks at the previous notes to predict the next note in the sequence based on the probabilities of sequences (n-grams) of tokens in the training data. These models can capture short-term dependencies but fail to model long-term coherence. This n-gram model is a significant improvement over random and frequency-based models because it captures local context/relationships without any deep learning or sequence modelling. Here, we are using a trigram model, where n=3.

Each of these models represents a different level of complexity and understanding of the melody structure. By comparing the GPT Melody model against these baselines, proper evaluations can be carried out on the model to understand whether it is learning useful features and structures beyond simple dependencies.

For evaluation of the GPT Model[2] and comparison with the above Baselines, the following techniques were used by comparing the generated melodies with the unseen validation data:

- **Edit Distance[4]:** This metric measures the minimum number of operations (insertions, deletions, substitutions) required to transform the generated melody into another validation data melody. It is useful when comparing how close a generated melody is to a reference. It captures structural differences but doesn't account for token frequency/similarity. *edit_distance()* function from the *ntlk* library has been used to evaluate the same. Here, lower value of edit distance is better as fewer changes are required to transform into ground truth.

```
from nltk.metrics.distance import edit_distance

def compute_edit_distance(seq1, seq2):
    return edit_distance(seq1, seq2)
```

- **Token Overlap:** This metrics measures the ratio of common tokens or notes between two sequences using Jaccard Similarity. This is a simple way to measure repetitive patterns in melodies however, it ignores the structure/order which is being captured above. Once, the melodies are converted to sets, removing duplicates, intersection and union of the sets are calculated and their ratio is returned. Here higher value of token overlap ratio (i.e. closer to 1) is better as it would mean that more notes are in common.

```
def compute_token_overlap(seq1, seq2):
    set1, set2 = set(seq1), set(seq2)
    intersection = len(set1 & set2)
    union = len(set1 | set2)
    return intersection / union
```

- **Sequence Similarity:** This metric combines both the above metrics to take weighted combinations of token overlap and edit distance, which is normalised to have values between 0 and 1. It helps incorporate the musical structure of the melody like rhythm, chord patterns etc. for a nuanced assessment. Here as well, a higher value of sequence similarity shows that the model captures the melody structure well.

```
def compute_sequence_similarity(seq1, seq2, weight=0.5):
    edit_dist = compute_edit_distance(seq1, seq2)
    token_overlap = compute_token_overlap(seq1, seq2)
    normalized_edit_distance = 1 - (edit_dist / max(len(seq1), len(seq2)))
    return weight * token_overlap + (1 - weight) * normalized_edit_distance
```

- **Perplexity[1]:** This metric measures how well the models predict the next token in a sequence modelling tasks such as melody generation in this case. It refers to the exponent of the average negative log-likelihood of the dataset. Here, lower value of perplexity generally indicated better performance of the GPT Model, as the model is more confident in its predictions. A good value of perplexity indicates that the GPT model understands the sequence patterns of the melodies like common transitions between notes and lower perplexity values show that the model finds the sequence more predictable and aligned with the patterns that the model learned. To calculate the same, for loop of batches of the selected melody is executed to create a list of losses in the model and the average of these losses is returned back. However, perplexity alone doesn't guarantee aesthetically pleasing or valid compositions. Perplexity can be calculated on both:
  - **Validation data** i.e. unseen real-world data. This provides a direct measure of how well the model has learned to generalize to unseen melody sequences.
  - **Generated data** i.e. sequences that the model generates. This helps assess how well the model maintains consistency and creativity in generated melody sequences.

```
def compute_perplexity(data):
        losses = []
        for x, y in zip(x_batches, y_batches):
                _, loss = model(x, y)
                losses.append(loss.item())
        avg_loss = sum(losses) / len(losses)
        return math.exp(avg_loss)
```

Now, executing these metrics on the existing GPT Model, we get the following results, shown in *Table 2*:

Models/Metrics	Edit Distance	Token Overlap	Sequence Similarity	Perplexity
GPT Model	**437**	0.6364	0.4815	Validation Data: 4.463 Generated Data: 6.027
Random Baseline Model	**424**	0.5	0.32	26.793
Frequency Baseline Model	**411**	0.5385	0.3642	22.321
N-Gram Baseline Model	**432**	0.5385	0.3492	10.797

Table 2: Evaluation Metrics of the GPT Model

From the given metrics, we can identify and compare the GPT Model for generating melodies with the various baseline models. Overall, the GPT model outperforms the baseline models, showcasing its ability to generate more realistic, consistent, and musically relevant melodies.

- In terms of edit distance, even though GPT Model has the highest value, this might indicate greater variety in its outputs. Whereas baseline models reflect their tendency to rely on repetitive or highly frequent patterns.
- In terms of token overlap, the GPT model achieves the highest token overlap(*0.6364*), showing that it aligns with the reference sequences better than the baselines. This indicates that the GPT model learns meaningful patterns from the training data.
- Also, in terms of sequence similarity, the GPT model achieves the highest sequence similarity(*0.4815*), showing its capability to learn and generate structural patterns in melodies, such as repetitions and progressions.
- From perplexity, the GPT model achieves significantly lower perplexity, both on validation and generated data, proving its ability to model sequences with long-term dependencies and higher-level patterns. The gap between validation (4.463) and generated data (6.027) perplexity suggests room for improvement in the generative capabilities of the model. Also, the Random and frequency-based baselines fail to sequence structure effectively which can be observed from their high perplexity values. The n-gram model, while being better than other baselines, is still significantly higher in perplexity values than the GPT Models.

Q. (2) Improve the model from that original architecture. Explain your reasoning when deciding what could be changed for improving the outcome. Include appropriate objective evaluation of the improvement. If deemed appropriate, subjective evaluation may also be included (a script for playing the output is included). Changes should involve improvements on the dataset (e.g., augmentation), on the architecture (with appropriate justifications), and on the training hyperparameters.

Now, to improve the existing GPT Model, the following changes were carried out:

- **Dataset Improvements:** Since the existing augmented dataset (*inputMelodiesAugmented.txt*) created from *augmentMidiTranslations.py* consists simply a 5-step pitch change of the notes dataset, further improvements are required to generalise the model better to unseen melodies. To perform these improvements, a separate python script was created (*augmentMidiTranslationsv2.py*) where the pitch shifts were changed from *[1, 2, 3, 4, 5]* to *[-2, -1, 0, 1, 2].* This was done to keep the melody pitch shifts around the existing melody and to also include the existing melody, which was not kept in the original code. Apart from this, noise was also added in the dataset in the form on random notes for 5% of the melodies. This will help simulate realistic imperfections in the real-world melodies. Post these improvements, the dataset was stored in *inputMelodiesAugmentedv2.txt* and used in the updated model.

```
def add_noise(melody):
       for note in melody:
           if random.random() < 0.05:
              noisy.append(random.choice(NOTES))
           noisy.append(note)
       return ''.join(noisy)
```

- **Architecture Improvements:** To improve the architecture of the existing GPT Model, changes targeting better learning and representation of melody data are introduced. Since melodies have two key components- pitch and time. Using separate embeddings for these aspects allows the model to clearly capture their relationships before combining them. This will help the model capture relationships within each in a better way and will allow the generation of diverse melodies that align with both pitch and time. Apart from this, relative positions of notes are often more important than their absolute positions in melodies and hence using this relative encoding will enhance the relationship understanding of the model. This will also help the model handle sequences of varying lengths better, as the relative positional information isn't linked to absolute melody length.

```
class GPTLanguageModel2(nn.Module):
   def __init__(self):
      super().__init__()
      self.pitch_embedding_table = nn.Embedding(vocab_size2, n_embd // 2)
      self.time_embedding_table = nn.Embedding(block_size, n_embd // 2)
      self.combined_projection = nn.Linear(n_embd, n_embd)
```

- **Hyperparameter Improvements:** Now, to improve on the existing hyperparameters, a grid search was applied on the hyperparameter values to identify the best combination for the GPT Model, based on estimated loss values. Here, a grid was created for a range of respective parameter values, also shown below for reference:

```
param_grid = {
   'learning_rate': [3e-4, 1e-3, 5e-4],
   'batch_size': [32, 64, 128],
   'dropout': [0.1, 0.2, 0.3],
   'n_layer': [4, 6, 8],
   'n_head': [4, 8, 16],
   'n_embd' : [512, 768]  }
```

Based on the loss scores, the Best Hyperparameters were : *{'learning_rate': 0.0005, 'batch_size': 128, 'dropout': 0.3, 'n_layer': 8, 'n_head': 8, 'n_embd': 768}* with the Best Validation Loss as *1.5014736652374268.* However, due to GPU resource constraints, selecting such high values of hyperparameters was not feasible and hence a moderate approach was taken with slightly higher values than the existing model, shown in *Table 3*. Increasing the *learning_rate* speeds up the training process by allowing the model to take larger steps towards the minimum of the loss function, while increasing the number of layers(*n_layer*) will allow the model to learn more complex patterns and representations in the melodies and increasing *dropout* will help prevent overfitting as it randomly deactivating neurons during training.

batch_size	block_size	max_iters	eval_interval	learning_rate	eval_iters	n_embd	n_head	n_layer	dropout
64	256	1000	500	5e-4	200	384	6	8	0.3

Table 3: Hyperparameters selected for the updated Model.

From the above three improvements on dataset, architecture and hyperparameters, the model was rerun, and the number of parameters came out to be `16.748558 M parameters`. Then, the model was trained on 1000 iterations, similar as before, and the loss values at the final step were `train loss 1.3937, val loss 1.3948.`

Here, we observe that the loss values are slightly higher than the original model which could be due to a number of reasons. Introducing changes such as relative positional encodings and separate embedding tables, which add complexity to the training process might require more epochs or finer hyperparameter tuning to show benefit and since we trained it for 1000 iterations only, it might not have enough time to fully converge due to increased complexity.

Further evaluations are also carried on the updated model considering the similar metrics as before. The output is shown in the *Table 4* below.

Models/Metrics	Edit Distance	Token Overlap	Sequence Similarity	Perplexity
Improved GPT Model	**425**	0.9286	0.5401	Validation Data: 4.579 Generated Data: 6.892

Table 4: Evaluation Metrics of the Improved GPT Model

- The improved GPT model has a better Edit Distance than the original GPT model (*425 < 437*), indicating that the improved model generates melodies closer to the references. Overall, still the frequency baseline performs better than all other models due to its reliance on common patterns but lacks flexibility for creative sequences.
- For token overlap, the improved GPT model achieves a significant improvement compared to the original GPT model (*0.9286 > 0.6364*). Here, the baselines perform poorly due to their inability to match tokens effectively across long sequences.
- In case of Sequence Similarity, the improved GPT model achieves the highest score showing that the improvements help it learn structural patterns of the melodies better. Even the original model was significantly better than the baselines but still lags behind the improved version as the baselines fail to capture any melody dependencies effectively.
- However, the perplexity score seems to have increased slightly for the improved GPT model compared to the original model which could be due to the added complexity like larger embedding size, positional encoding, dropout, etc., which sometimes makes optimisation harder.

Despite the increase in perplexity, the improved model still performs better on Edit Distance, Token Overlap, and Sequence Similarity, indicating that melodies are more meaningful and aligned with melody dataset even if prediction uncertainty has slightly increased.

Apart for quantitative comparison, qualitative/subjective analysis was also performed. Using the code piece given in *melodyPlay.py,* the generated melodies from both the original and updated GPT models as well as the baseline models were extracted in *.wav* format. Upon listening to the GPT melodies, one can identify rhythms as well as patterns in these melodies whereas the baseline melodies lack in these aspects. Also, when comparing both the GPT Melodies, the updated GPT Model's melody is clearly more rhythmic and pleasing to listen to. Also, the updated melody feels unique with no abrupt changes in notes. All these melodies have been provided in the ZIP file along with the code.

PART 2

Q. (i) What is an ROC curve? How can it be used to evaluate the performance of a classifier compared with a baseline classifier? Why would you use an ROC curve instead of a classification accuracy metric?

An ROC Curve is a graphical plot created by plotting the true positive rate (TPR) against the false positive rate (FPR). It helps us evaluate the performance of binary classification models. Below are the formulas for TPR and FPR respectively, where TP is True Positives, FN is False Negatives, FP is False Positives, and TN is True Negatives.

$$TPR = TP/(TP + FN) \qquad FPR = FP/(FP + TN)$$

Typically, a baseline classifier might be a random guess or a simple model like predicting the majority class. On an ROC curve, a random classifier would produce a diagonal line from the bottom left to the top right (FPR = TPR). A classifier with better performance will produce a curve that is above this diagonal, showing higher TPR for the same FPR. The further the curve is from the diagonal line towards the top-left corner, the better the classifier performs. Also, the area under the curve (AUC) measures this and a larger AUC shows better performance of the classifier. An ROC Curve will be preferred over the classification accuracy metric when:

- Dataset is imbalanced and the classifier is biased towards the majority class. ROC curves are less sensitive to such imbalances.
- Comparison of different classifiers is required as the curves can be visually compared along with the AUC Values.
- Complete view of different thresholds is required for different classifications. This is not possible in case of accuracy as it is dependent on the threshold.

Q. (ii) Give two examples of situations where a linear regression would give inaccurate predictions. Explain your reasoning and what possible solutions you would adopt in each situation.

a) **Existence of outliers** in the dataset is a situation where linear regression would give inaccurate predictions. Linear regression is sensitive to outliers because it minimizes the sum of squared residuals, which can be influenced by extreme values skewing the linear model. This results in a regression line that might not represent the general trend well. Outliers generally have large difference between the predicted and actual values. This increases the overall error of the model, even if the fit is good for the majority of the data. Some possible solutions to this could be:
   - Using diagnostic plots (like residual plots) to identify potential outliers.
   - Carefully identify and remove the outliers after determining whether they are data entry errors or truly exceptional cases.
   - Statistical methods like the Z-score can also be used to identify and potentially remove outliers.
   - Transforming the response variable can also reduce the effect of outliers by bringing extreme values closer to the bulk of the data.

b) **Non-Linearity** in the relationships of the variables is also a situation where linear regression would give inaccurate predictions. Linear regression usually assumes a straight-line relationship between X and Y and if the data shows a curve (like an exponential or logarithmic relationship), linear regression will fit a line that does not capture this curvature, leading to underfitting of the dataset. Some possible solutions to this could be:
   - Applying a mathematical transformation to the dependant or independent variables of the dataset.
   - Using polynomial regression to fit the curve as this would add the polynomial terms to the model and capture the nonlinear pattern.
   - Use models like Decision Trees, Random Forests, or neural networks.

Q. (iii) The term 'kernel' has different meanings in SVM and CNN models. Explain the two different meanings. Discuss why and when the use of SVM kernels and CNN kernels is useful, as well as mentioning different types of kernels.

In SVMs, a kernel function is a method used to transform data into a higher-dimensional space where a linear decision boundary can separate classes that are not linearly separable in the original feature space known as kernel-trick. Whereas, in CNNs, the term "kernel" refers to a small matrix or filter that slides over the input data (like an image) to perform convolution. These operations extract features from the input data, such as edges, corners, and textures.

For SVMs, kernels are particularly useful when dealing with data that isn't linearly separable in its original form. This kernel trick allows for calculation in high-dimensional space without the need for transforming the data. Below are the types of SVM Kernels:

- Linear Kernel – Used when data is linearly separable and performs a dot product between two vectors.
- Polynomial Kernel – Used when data has polynomial relationships as it can capture nonlinear patterns.
- Radial Basis Function Kernel – Used when there's no prior knowledge of the data as it maps data to an infinite-dimensional space.

For CNNs, kernels are crucial for detecting local patterns like edges, textures, or colours in images. The same kernel is applied to multiple locations in the input image, reducing the number of parameters and improving efficiency. Below are the types of CNN Kernels:

- Square Kernels - Most common kernels and used for spatial hierarchies in images.
- 1D Kernels - Applied in time series or text data for sequential feature extraction.
- Sobel kernel - Used for edge detection.

Q. (iv) In k-fold cross-validation, a dataset is resampled multiple times. What is the idea behind this resampling i.e. why does resampling allow us to evaluate the generalisation performance of a machine learning model. Give a small example to illustrate. Discuss when it is and it is not appropriate to use k-fold cross-validation.

In k-fold cross-validation, the dataset is divided into 'k' equal-sized subsets. It helps to provide a more accurate estimate of how well a machine learning model will generalize to an independent dataset.

- By dividing the dataset into k parts, training the model k times on different subsets, and validating it on the remaining part, k different performance metrics are fetched for the model.
- In each iteration, a different fold is used for validation, ensuring that every data point has a chance to be in the validation set.
- It mimics the process of training on one dataset and testing on another.
- Cross-validation assists in selecting the best model among multiple candidates by comparing their performance across different folds.

For example, there is a dataset of 100 sample values and k is chosen to be 5. First the data is divided into 5 folds i.e. 1-20, 21-40 and so on. First iteration would select 1st fold as the validation data and remaining 4 folds as the training data. In the next iteration, 2nd fold is selected as the validation data and remaining as training data. This is continued till all folds are used as validation dataset. Post this, evaluation is carried on each iteration and performance metrics are computed. Average of these metrics is the cross-validation score.

Usually, k-fold is used when:

- There is a small dataset.
- There are multiple models to select from.
- Model's consistency of performance needs to be checked.

Also, k-fold validation is not used when:

- There is a very large dataset which may result in high computational cost.
- When order of data matters i.e. time series data.
- Data is imbalanced as the folds will also be imbalanced.
- Data is of high dimensions as it will cause high variance.

3 References

[1] https://en.wikipedia.org/wiki/Perplexity

[2] https://ssahuupgrad-93226.medium.com/llm-evaluation-metrics-the-ultimate-llm-evaluation-guide-e9bc94dba1e1

[3] https://web.stanford.edu/~jurafsky/slp3/3.pdf

[4] https://en.wikipedia.org/wiki/Edit_distance

4 Appendix

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import torch
import torch.nn as nn
from torch.nn import functional as F

# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 1000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
# ------------

print('Hyperparameters Updated')

torch.manual_seed(1337)

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

# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
print('Characters: ',chars)
vocab_size = len(chars)
print('Vocabulary Size: ',vocab_size)
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string

# Train and test splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

```
# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y

@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out

class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B, hs, T) -> (B, T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

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

class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)
    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class GPTLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

        # better init, not covered in the original GPT video, but important, will cover in followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # 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

model = GPTLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    print('Iteration: ',iter)
    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

torch.save(model.state_dict(), 'model_2.pth')
'''# Recreate the model architecture
loaded_model = GPTLanguageModel()
loaded_model.load_state_dict(torch.load('model_1.pth'))
loaded_model.to(device)
loaded_model.eval()
print("Model loaded from gpt_music_model.pth")'''

context = torch.zeros((1, 1), dtype=torch.long, device=device)
gpt_melody = decode(m.generate(context, max_new_tokens=500)[0].tolist())
print(gpt_melody)

import math

def compute_perplexity(melody, model):
    tokenized = [stoi[ch] for ch in melody if ch in stoi]

    if len(tokenized) < block_size:
        raise ValueError("Melody length must be greater than or equal to block_size.")

    x_batches = []
    y_batches = []
    for i in range(0, len(tokenized) - block_size):
        x = tokenized[i:i + block_size]
        y = tokenized[i + 1:i + 1 + block_size]
        x_batches.append(torch.tensor(x, dtype=torch.long))
        y_batches.append(torch.tensor(y, dtype=torch.long))

    model.to(device)
    model.eval()

    losses = []
    with torch.no_grad():
        for x, y in zip(x_batches, y_batches):
            x, y = x.unsqueeze(0).to(device), y.unsqueeze(0).to(device)
            _, loss = model(x, y)
            losses.append(loss.item())

    avg_loss = sum(losses) / len(losses)
    perplexity = math.exp(avg_loss)
    return perplexity

import random

def generate_random_melody(length, vocab):
    return ''.join(random.choices(vocab, k=length))

random_melody = generate_random_melody(500, chars)
print("Random Melodies:")
print(random_melody)

from collections import Counter
```

```
def compute_token_frequencies(data, chars):
    counts = Counter(data)
    total = sum(counts.values())
    probabilities = {char: counts[char] / total for char in chars}
    return probabilities

freq_probs = compute_token_frequencies(list(text), chars)
print("Frequencies:", freq_probs)
def generate_freq_based_melody(length, freq_probs):
    vocab, probs = zip(*freq_probs.items())
    return ''.join(random.choices(vocab, probs, k=length))

freq_melody = generate_freq_based_melody(500, freq_probs)
print("\nFrequency-Based Melodies:")
print(freq_melody)

from collections import defaultdict

def build_ngram_model(data, n):
    ngrams = defaultdict(Counter)
    for i in range(len(data) - n):
        context = tuple(data[i:i+n-1])
        next_token = data[i+n-1]
        ngrams[context][next_token] += 1

    ngram_probs = {context: {token: count / sum(counter.values())
                            for token, count in counter.items()}
                   for context, counter in ngrams.items()}
    return ngram_probs

ngram_model = build_ngram_model(list(text), n=3)

def generate_ngram_melody(length, ngram_model, vocab, n):
    melody = [random.choice(vocab) for _ in range(n-1)]
    for _ in range(length - (n-1)):
        context = tuple(melody[-(n-1):])
        next_token = random.choices(
            list(ngram_model[context].keys()),
            weights=list(ngram_model[context].values()),
            k=1
        )[0] if context in ngram_model else random.choice(vocab)
        melody.append(next_token)
    return ''.join(melody)

ngram_melody = generate_ngram_melody(500, ngram_model, chars, n=3)
print("N-Gram Melody:", ngram_melody)
```

```
gpt_perplexity = compute_perplexity(gpt_melody, model)
random_perplexity = compute_perplexity(random_melody, model)
freq_perplexity = compute_perplexity(freq_melody, model)
ngram_perplexity = compute_perplexity(ngram_melody, model)

print(f"GPT Model Perplexity: {gpt_perplexity}")
print(f"Random Baseline Model Perplexity: {random_perplexity}")
print(f"Frequency Based Baseline Model Perplexity: {freq_perplexity}")
print(f"N-Gram Baseline Model Perplexity: {ngram_perplexity}")

sample_val_melody = decode(val_data[:500].tolist())
print(sample_val_melody)

gpt_val_perplexity = compute_perplexity(sample_val_melody, model)
print(f"GPT Model Perplexity on Validation Data: {gpt_val_perplexity}")

from nltk.metrics.distance import edit_distance

def compute_edit_distance(seq1, seq2):
    return edit_distance(seq1, seq2)

def compute_token_overlap(seq1, seq2):
    set1, set2 = set(seq1), set(seq2)
    intersection = len(set1 & set2)
    union = len(set1 | set2)
    return intersection / union

def compute_sequence_similarity(seq1, seq2, weight=0.5):
    edit_dist = compute_edit_distance(seq1, seq2)
    token_overlap = compute_token_overlap(seq1, seq2)
    normalized_edit_distance = 1 - (edit_dist / max(len(seq1), len(seq2)))
    return weight * token_overlap + (1 - weight) * normalized_edit_distance

models = {
    "GPT": gpt_melody,
    "Random": random_melody,
    "Frequency": freq_melody,
    "N-gram": ngram_melody
}

results = {}

for model_name, melody in models.items():
    edit_dist = compute_edit_distance(melody, sample_val_melody)
    token_overlap = compute_token_overlap(melody, sample_val_melody)
    seq_similarity = compute_sequence_similarity(melody, sample_val_melody)
    results[model_name] = {
        "Edit Distance": edit_dist,
        "Token Overlap": token_overlap,
        "Sequence Similarity": seq_similarity
    }

# Display Results
for model, metrics in results.items():
    print(f"Model: {model}")
    for metric, value in metrics.items():
        print(f"  {metric}: {value:.4f}")

pip install simpleaudio

from pydub import AudioSegment
import numpy as np
import simpleaudio as sa



NOTE_FREQUENCIES = {
    'C': 261.63,
    'c': 277.18,  # C#
    'D': 293.66,
    'd': 311.13,  # D#
    'E': 329.63,
    'F': 349.23,
    'f': 369.99,  # F#
    'G': 392.00,
    'g': 415.30,  # G#
    'A': 440.00,
    'a': 466.16,  # A#
    'B': 493.88,
    'R': 0     # Rest
}



# Generate a sine wave for a given frequency
def generate_sine_wave(frequency, duration_ms, sample_rate=44100, amplitude=0.5):
    t = np.linspace(0, duration_ms / 1000, int(sample_rate * duration_ms / 1000), False)
    wave = 0.5 * amplitude * np.sin(2 * np.pi * frequency * t)
    wave = (wave * 32767).astype(np.int16)
    audio_segment = AudioSegment(
        wave.tobytes(),
        frame_rate=sample_rate,
        sample_width=wave.dtype.itemsize,
        channels=1
    )
    return audio_segment
```

```
# Function to create a sequence of notes
def create_sequence(note_sequence, duration_ms=500):
    song = AudioSegment.silent(duration=0)
    for note in note_sequence:
        if note == 'R':  # Handle rest
            segment = AudioSegment.silent(duration=duration_ms)
        else:
            frequency = NOTE_FREQUENCIES[note]
            segment = generate_sine_wave(frequency, duration_ms)
        song += segment
    return song

song = create_sequence(gpt_melody.replace('\n', ''), duration_ms=500)  # 500ms per note

# Save the song to a .wav file
song.export("gpt_melody.wav", format="wav")

# Play the .wav file using simpleaudio
'''wave_obj = sa.WaveObject.from_wave_file("gpt_melody.wav")
play_obj = wave_obj.play()
play_obj.wait_done()'''

song = create_sequence(random_melody.replace('\n', ''), duration_ms=500)  # 500ms per note

# Save the song to a .wav file
song.export("random_melody.wav", format="wav")

# Play the .wav file using simpleaudio
'''wave_obj = sa.WaveObject.from_wave_file("random_melody.wav")
play_obj = wave_obj.play()
play_obj.wait_done()'''

song = create_sequence(freq_melody.replace('\n', ''), duration_ms=500)  # 500ms per note

# Save the song to a .wav file
song.export("freq_melody.wav", format="wav")

# Play the .wav file using simpleaudio
'''wave_obj = sa.WaveObject.from_wave_file("freq_melody.wav")
play_obj = wave_obj.play()
play_obj.wait_done()'''

song = create_sequence(ngram_melody.replace('\n', ''), duration_ms=500)  # 500ms per note

# Save the song to a .wav file
song.export("ngram_melody.wav", format="wav")
```

```
# Play the .wav file using simpleaudio
'''wave_obj = sa.WaveObject.from_wave_file("ngram_melody.wav")
play_obj = wave_obj.play()
play_obj.wait_done()'''

from itertools import product

def grid_search(param_grid):

  best_params = {}
  best_val_loss = float('inf')

  for params in product(*param_grid.values()):

    global learning_rate, batch_size, dropout, n_layer, n_head
    learning_rate = params[0]
    batch_size = params[1]
    dropout = params[2]
    n_layer = params[3]
    n_head = params[4]
    n_embd = params[5]



    print('learning_rate=',learning_rate)
    print('batch_size=',batch_size)
    print('dropout=',dropout)
    print('n_layer=',n_layer)
    print('n_head=',n_head)
    print('n_embd=',n_embd)

    model2 = GPTLanguageModel2()
    m2 = model2.to(device)

    print(sum(p.numel() for p in m2.parameters())/1e6, 'M parameters')
    optimizer = torch.optim.AdamW(model2.parameters(), lr=learning_rate)

    val_loss = estimate_loss2()['val']
    print(val_loss)

    if val_loss < best_val_loss:
      best_val_loss = val_loss
      best_params = dict(zip(param_grid.keys(), params))

  return best_params, best_val_loss

# Define the hyperparameter grid
```

```
param_grid = {
    'learning_rate': [3e-4, 1e-3, 5e-4],
    'batch_size': [32, 64, 128],
    'dropout': [0.1, 0.2, 0.3],
    'n_layer': [4, 6, 8],
    'n_head': [4, 8, 16],
    'n_embd' : [512, 768]  }



best_params, best_val_loss = grid_search(param_grid)
print(f"Best Hyperparameters: {best_params}")
print(f"Best Validation Loss: {best_val_loss}")

import torch
import torch.nn as nn
from torch.nn import functional as F

# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 1000
eval_interval = 500
learning_rate = 5e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 8
dropout = 0.3
# ------------

print('Hyperparameters Updated')

with open('inputMelodiesAugmentedv2.txt', 'r', encoding='utf-8') as f2:
    text2 = f2.read()

# here are all the unique characters that occur in this text
chars2 = sorted(list(set(text2)))
print('Characters: ',chars2)
vocab_size2 = len(chars2)
print('Vocabulary Size: ',vocab_size2)
# create a mapping from characters to integers
stoi2 = { ch:i for i,ch in enumerate(chars2) }
itos2 = { i:ch for i,ch in enumerate(chars2) }
encode2 = lambda s: [stoi2[c] for c in s] # encoder: take a string, output a list of integers
decode2 = lambda l: ''.join([itos2[i] for i in l]) # decoder: take a list of integers, output a string
```

```
# Train and test splits
data2 = torch.tensor(encode2(text2), dtype=torch.long)
n = int(0.9*len(data2)) # first 90% will be train, rest val
train_data2 = data2[:n]
val_data2 = data2[n:]

# data loading
def get_batch2(split):
    # generate a small batch of data of inputs x and targets y
    data2 = train_data2 if split == 'train' else val_data2
    ix = torch.randint(len(data2) - block_size, (batch_size,))
    x = torch.stack([data2[i:i+block_size] for i in ix])
    y = torch.stack([data2[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y

@torch.no_grad()
def estimate_loss2():
    out = {}
    model2.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch2(split)
            logits, loss = model2(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model2.train()
    return out



class Head2(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.relative_positions = nn.Embedding(2 * block_size - 1, head_size)

        self.dropout = nn.Dropout(dropout)
        self.pos_emb = nn.Parameter(torch.randn(block_size, head_size))
        #self.tril = torch.tril(torch.ones(block_size, block_size))
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)



         # Compute relative positional embeddings
        pos_emb = self.pos_emb[:T, :].unsqueeze(0)  # (1, T, head_size)
        relative_wei = torch.einsum('bth,tlh->btl', q, pos_emb)  # (B, T, T)

        # Standard self-attention with relative positional encoding
        wei = (q @ k.transpose(-2, -1) + relative_wei) * (C ** -0.5)  # (B, T, T)
        #wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))  # Apply causal masking
        wei = wei.masked_fill(self.tril[:T, :T].to(x.device) == 0, float('-inf'))  # Apply causal masking




        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)



        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

class MultiHeadAttention2(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head2(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedFoward2(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

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

class Block2(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention2(n_head, head_size)
        self.ffwd = FeedFoward2(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class GPTLanguageModel2(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.pitch_embedding_table = nn.Embedding(vocab_size2, n_embd // 2)
        self.time_embedding_table = nn.Embedding(block_size, n_embd // 2)
        self.combined_projection = nn.Linear(n_embd, n_embd)

        self.blocks = nn.Sequential(*[Block2(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size2)

        # better init, not covered in the original GPT video, but important, will cover in followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        pitch_emb = self.pitch_embedding_table(idx)  # (B, T, n_embd//2)
        time_emb = self.time_embedding_table(torch.arange(T, device=device))  # (T, n_embd//2)

        # Combine embeddings
        combined_emb = torch.cat((pitch_emb, time_emb.unsqueeze(0).expand(B, -1, -1)), dim=-1)
        x = self.combined_projection(combined_emb)

        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # 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

model2 = GPTLanguageModel2()
```

```
m2 = model2.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m2.parameters())/1e6, 'M parameters')

optimizer = torch.optim.AdamW(model2.parameters(), lr=learning_rate)

for iter in range(max_iters):
    print('Iteration: ',iter)
    # every once in a while evaluate the loss on train and val sets
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss2()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

    # sample a batch of data
    xb, yb = get_batch2('train')

    # evaluate the loss
    logits, loss = model2(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

torch.save(model2.state_dict(), 'model_5.pth')

context = torch.zeros((1, 1), dtype=torch.long, device=device)
gpt_melody2 = decode2(m2.generate(context, max_new_tokens=500)[0].tolist())
print(gpt_melody2)

def compute_perplexity2(melody, model):
    tokenized = [stoi2[ch] for ch in melody if ch in stoi2]

    if len(tokenized) < block_size:
        raise ValueError("Melody length must be greater than or equal to block_size.")

    x_batches = []
    y_batches = []
    for i in range(0, len(tokenized) - block_size):
        x = tokenized[i:i + block_size]
        y = tokenized[i + 1:i + 1 + block_size]
        x_batches.append(torch.tensor(x, dtype=torch.long))
        y_batches.append(torch.tensor(y, dtype=torch.long))

    model.to(device)
    model.eval()

    losses = []
    with torch.no_grad():
        for x, y in zip(x_batches, y_batches):
            x, y = x.unsqueeze(0).to(device), y.unsqueeze(0).to(device)  # Add batch dim
            _, loss = model(x, y)
            losses.append(loss.item())

    avg_loss = sum(losses) / len(losses)
    perplexity = math.exp(avg_loss)
    return perplexity

sample_val_melody2 = decode2(val_data2[500:1000].tolist())
print(sample_val_melody2)

gpt_val_perplexity_2 = compute_perplexity(sample_val_melody2, model2)
print(f"GPT Model Perplexity on Validation Data: {gpt_val_perplexity_2}")
gpt_perplexity_2 = compute_perplexity(gpt_melody2, model2)
print(f"GPT Model Perplexity on Validation Data: {gpt_perplexity_2}")

models = {
    "GPT": gpt_melody2
}

results = {}

for model_name, melody in models.items():
    edit_dist = compute_edit_distance(melody, sample_val_melody2)
    token_overlap = compute_token_overlap(melody, sample_val_melody2)
    seq_similarity = compute_sequence_similarity(melody, sample_val_melody2)
    results[model_name] = {
        "Edit Distance": edit_dist,
        "Token Overlap": token_overlap,
        "Sequence Similarity": seq_similarity
    }

# Display Results
for model, metrics in results.items():
    print(f"Model: {model}")
    for metric, value in metrics.items():
        print(f"  {metric}: {value:.4f}")

song = create_sequence(gpt_melody2.replace('\n', ''), duration_ms=500)  # 500ms per note

# Save the song to a .wav file
song.export("gpt_melody_updated.wav", format="wav")
```