CSU44061 Machine Learning - Final Project

Faith Olopade - 21364066

Appendix contains all the code.

Part 1

Task 1

To adapt the given GPT.py text generation code for melody generation, I utilised the inputMelodiesAugmented.txt dataset, which encodes simplified melodies into a single-octave token vocabulary. This approach minimises the complexity while preserving the fundamental structure of melodies, aligning with the hints provided.

Minimal changes were made to the original code, focusing on replacing the text vocabulary with the melody token set. Specifically:

1. Tokenization Adjustments: Instead of handling characters from text, I modified the tokenization process to encode the musical notes (C, C#, D, ...) and rests (R) into numerical representations, ensuring that each token corresponds to a unique integer.
2. Dataset Handling: The melody dataset was split into training (90%) and validation (10%) sets to facilitate evaluation. This ensures the model can generalise beyond the training data.

```
Step 0: train loss 2.6962, val loss 2.6865
Baseline accuracy: 0.0780
Step 500: train loss 1.6796, val loss 1.6102
Baseline accuracy: 0.0780
Step 1000: train loss 1.3363, val loss 1.3167
Baseline accuracy: 0.0780
Step 1500: train loss 1.1942, val loss 1.2047
Baseline accuracy: 0.0780
Step 2000: train loss 1.1087, val loss 1.1500
Baseline accuracy: 0.0780
Step 2500: train loss 1.0230, val loss 1.1508
Baseline accuracy: 0.0780
Step 3000: train loss 0.9260, val loss 1.1711
Baseline accuracy: 0.0780
Step 3500: train loss 0.8073, val loss 1.2351
Baseline accuracy: 0.0780
Step 4000: train loss 0.6774, val loss 1.3119
Baseline accuracy: 0.0780
Step 4500: train loss 0.5621, val loss 1.4189
Baseline accuracy: 0.0780
Step 4999: train loss 0.4624, val loss 1.5578
Baseline accuracy: 0.0780

FCaCCdCFgFdFgFRFFCaCCFddCaCagCCdCagFFRFCaCCdCdCagFFRFFggFgFaCCdCdCaggFFRFFggFgFaCCdCdCagFFRFFggFgFaC
```

Figure 1: Model Output for Melody Generation

Training Process

The training used the same architecture as the original GPT model, with positional embeddings and transformer blocks repurposed to handle sequences of musical tokens. During training:

- The loss steadily decreased from an initial 2.6962 to 0.4624 at step 4999 for training data, as shown in Figure 1. The validation loss followed a similar trend initially, reaching 1.5578 at the final step, indicating overfitting.
- A baseline accuracy of 0.0780 was used, calculated as the chance of predicting the most frequent token. This served as a reference point to evaluate model performance.

Evaluation Metrics

The main evaluation metrics were:

1. Cross-Entropy Loss: This quantified how well the model predicted the next token in the sequence, with lower values indicating better performance.
2. Baseline Accuracy: This was essential to contextualise the model's predictive power, given the simplified nature of the dataset.

Observations from Training

- General Trends: Training loss consistently decreased, demonstrating the model's ability to learn the sequence structure of melodies. However, validation loss diverged after step 2500, signaling potential overfitting.

- Challenges: Overfitting arose because of the small vocabulary and repetitive patterns in the dataset. To mitigate this, dropout regularisation (0.2) was applied in attention and feedforward layers.

Generated Output

Figure 1 shows a sample of the generated melody:
FCaCCdCFgFdFgFRFFCaCCFddCaCagCCdCagFFRFCaCCdCdCagFFRFFggFgFaCCdCdCaggFFRFFggFgFaCCdCdCagFFRFFggFgFaC

This sequence demonstrates coherent melodic patterns within the constraints of the single-octave vocabulary. While repetitive in places, the generated melodies align with the training data's structure.

Task 2

Dataset Improvements

I expanded the provided inputMelodiesAugmented.txt dataset by introducing additional pitch variations and subtle noise. This was achieved using a custom augment_dataset() function, which applied random transpositions to the input melodies. By augmenting the dataset, the model was exposed to a more diverse set of training samples, improving its generalisation ability. This step also aligns with the hints that suggested exploring the melody's pitch and time dimensions.

Architectural Adjustments

I modified the original architecture to better capture the nuances of melodic patterns. Specifically, I:

1. Increased the embedding dimension (n_embd) from 384 to 512, providing the model with a richer feature space.
2. Added more layers (n_layer = 8) to the transformer, enhancing its capacity to model complex temporal relationships.
3. Raised the number of attention heads (n_head = 8) to improve the model's ability to focus on multiple aspects of the input simultaneously.

These changes were informed by the need for the model to process sequential musical data effectively while maintaining efficiency in computation.

Hyperparameter Tuning

To stabilise training and achieve better convergence:

- I reduced the learning rate to 1e-4 to allow finer updates during training.
- The dropout rate was increased to 0.3 to mitigate overfitting, particularly due to the larger model capacity introduced by the architectural changes.

Evaluation

```
Step 0: train loss 3.8433, val loss 3.7903
Step 500: train loss 2.3242, val loss 2.5031
Step 1000: train loss 2.2938, val loss 2.4877
Step 1500: train loss 2.2854, val loss 2.4818
Step 2000: train loss 2.2663, val loss 2.4789
Step 2500: train loss 2.2622, val loss 2.4740
Step 3000: train loss 2.2438, val loss 2.4706
Step 3500: train loss 2.2420, val loss 2.4699
Step 4000: train loss 2.2173, val loss 2.4669
Step 4500: train loss 2.1847, val loss 2.4622
Step 4999: train loss 2.1662, val loss 2.4662

ddGRBcaBfaRafafdfEcddcBAFgGfggBEcAc

Gf
RRacRcdddGCdcdARAcdAg
GfCEggcfafEgddggAcFAcfgagdfgGaAfaFaDaa
```

Figure 2: Model Improvement Output

The model's performance was monitored using training and validation loss values across iterations (Figure 2). The results indicated a consistent decrease in both metrics, confirming effective learning:

- Initial loss: Train = 3.8433, Val = 3.7903
- Final loss: Train = 2.1662, Val = 2.4662

This significant reduction, particularly in the validation loss, suggests that the model generalises well to unseen data.

The generated melodies displayed a noticeable improvement in coherence and diversity compared to the baseline outputs. Listening tests revealed that the augmented dataset and enhanced architecture allowed the model to produce sequences with smoother transitions and more musically interesting patterns.

Part 2

(i)

An ROC (Receiver Operating Characteristic) curve plots the true positive rate (TPR) against the false positive rate (FPR) at different classification thresholds. It helps evaluate a classifier by showing its ability to distinguish between classes. The area under the ROC curve (AUC) quantifies this performance, a higher AUC means better discrimination.

Using an ROC curve is better than accuracy in imbalanced datasets because accuracy might be misleading if one class dominates. For example, a model that predicts the majority class always will have high accuracy but perform poorly in distinguishing between classes. An ROC curve avoids this by evaluating performance over a range of thresholds.

(ii)

Non-linear relationships: Linear regression assumes a straight-line relationship between variables. For example, predicting temperature based on time of day, where the relationship follows a curve.

- Solution: Add polynomial features to capture the curve or use models like decision trees or neural networks better suited to non-linear data.

High collinearity: When predictors are highly correlated, the model struggles to determine the individual contribution of each predictor. For instance, if TV and radio advertising budgets are highly correlated.

- Solution: Use Ridge or Lasso regression to regularise the model and reduce multicollinearity.

(iii)

In SVMs, a kernel transforms data into a higher-dimensional space to make it linearly separable. Common kernels include linear, polynomial, and radial basis function (RBF). Kernels are useful when the data is not linearly separable in its original space, like classifying circular patterns using an RBF kernel.

In CNNs, a kernel is a small matrix used in convolution operations to detect patterns (e.g., edges in images). Kernels slide over the input image, performing element-wise multiplication to extract features. They are crucial for feature detection and pattern recognition in image data.

(iv)

In k-fold cross-validation, the dataset is split into k subsets. Each subset is used as test data while the rest are used for training, ensuring every point is tested. This approach evaluates the model's performance on unseen data and reduces variance compared to a single train-test split.

- Example: For 5-fold cross-validation, if a dataset has 100 points, it is split into 5 groups of 20. The model trains on 80 points and tests on 20, repeating this process 5 times.

It’s appropriate when data is limited or when a robust evaluation is needed. It is less suitable for time-series data, where data points are dependent, or for large datasets where computational cost is high.

Appendix

Part 1

Task 1

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

# Hyperparameters
batch_size = 64  # Number of sequences to process in parallel
block_size = 256  # Maximum context length for predictions
max_iters = 5000
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

torch.manual_seed(1337)

# Load melody data
with
open('/kaggle/input/inputmelodiesaugmented-txt/inputMelodiesAugmented.
txt', 'r', encoding='utf-8') as f:
    text = f.read()

# Tokenization
chars = sorted(list(set(text)))
vocab_size = len(chars)
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]
decode = lambda l: ''.join([itos[i] for i in l])

# Prepare data
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9 * len(data))
```

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

# Data batching
def get_batch(split):
    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])
    return x.to(device), y.to(device)

@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

# Baseline: Predicts the next token as the most frequent token
@torch.no_grad()
def baseline_accuracy():
    target_counts = torch.bincount(train_data)
    most_common = torch.argmax(target_counts).item()
    baseline_predictions = torch.full_like(val_data[:-1],
fill_value=most_common)
    accuracy = (baseline_predictions ==
val_data[1:]).float().mean().item()
    return accuracy

# GPT Model
class GPTLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        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) for _ in
range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)
        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
        tok_emb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(torch.arange(T,
device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)

        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):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
```

```
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)
        return idx

class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(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 MultiHeadAttention(nn.Module):
    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)
        return self.dropout(self.proj(out))

class Head(nn.Module):
    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):
        B, T, C = x.shape
```

```
        k = self.key(x)
        q = self.query(x)
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        return wei @ v

class FeedForward(nn.Module):
    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)

# Training
model = GPTLanguageModel().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    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}")
        print(f"Baseline accuracy: {baseline_accuracy():.4f}")

    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Generate melodies
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(model.generate(context, max_new_tokens=100)[0].tolist()))
```

Task 2

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

# Hyperparameters
batch_size = 64
block_size = 256
max_iters = 5000
eval_interval = 500
learning_rate = 1e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 512
n_head = 8
n_layer = 8
dropout = 0.3

torch.manual_seed(1337)

# Load and preprocess the melody dataset
with
open('/kaggle/input/inputmelodiesaugmented-txt/inputMelodiesAugmented.
txt', 'r', encoding='utf-8') as f:
    text = f.read()

# Tokenization
chars = sorted(list(set(text)))
vocab_size = len(chars)
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for ch, i in stoi.items()}
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])

# Dataset augmentation: Introduced random noise and transpose
variations
def augment_dataset(data, vocab_size, num_augmentations=3):
```

```
    augmented_data = [data.clone()]  # Start with a list containing
the original data
    for _ in range(num_augmentations):
        # Random transpose
        transposed = [(note + random.randint(-2, 2)) % vocab_size for
note in data.tolist()]
        augmented_data.append(torch.tensor(transposed,
dtype=torch.long))  # Append the new tensor
    return torch.cat(augmented_data)  # Concatenate all tensors into
one


data = torch.tensor(encode(text), dtype=torch.long)
data = augment_dataset(data, vocab_size)
n = int(0.9 * len(data))
train_data = data[:n]
val_data = data[n:]

# Data batching
def get_batch(split):
    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])
    return x.to(device), y.to(device)

@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().item()
    model.train()
    return out

# Model definition
class GPTMelodyModel(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        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) for _ in
range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.xavier_uniform_(module.weight)
            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
        tok_emb = self.token_embedding_table(idx)
        pos_emb = self.position_embedding_table(torch.arange(T,
device=device))
        x = tok_emb + pos_emb
        x = self.blocks(x)
        x = self.ln_f(x)
        logits = self.lm_head(x)

        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):
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]
```

```
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
            idx = torch.cat((idx, idx_next), dim=1)
        return idx

class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(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 MultiHeadAttention(nn.Module):
    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)
        return self.dropout(self.proj(out))

class Head(nn.Module):
    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):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)
        wei = q @ k.transpose(-2, -1) * (k.shape[-1] ** -0.5)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        return wei @ v

class FeedForward(nn.Module):
    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)

# Training
model = GPTMelodyModel().to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    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}")

    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Generate melodies
```

```
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(model.generate(context, max_new_tokens=100)[0].tolist()))
```

Convert Generated Melodies to MIDI

```
# -*- coding: utf-8 -*-
"""
@author: Giovanni Di Liberto (I cleaned it up a bit just removed the parts
that weren't relevant to getting MIDI file)
See description in the assignment instructions.
"""

import os
from mido import MidiFile, MidiTrack, Message

# Define the note dictionary
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,
}

# Map MIDI note numbers to note names (ignoring octaves)
MIDI_NOTE_TO_NAME = {
    0: 'C', 1: 'c', 2: 'D', 3: 'd', 4: 'E', 5: 'F', 6: 'f', 7: 'G', 8:
'g', 9: 'A', 10: 'a', 11: 'B'
```

```
}

# Function to convert text sequence to MIDI
def text_sequence_to_midi(sequence, output_path):
    midi = MidiFile()
    track = MidiTrack()
    midi.tracks.append(track)

    # Remove spaces from the sequence
    sequence = sequence.replace(' ', '')

    for note in sequence:
        if note == 'R':
            # Add a rest (note_off with some duration)
            track.append(Message('note_off', note=0, velocity=0,
time=480))
        else:
            # Map the note to a MIDI pitch
            midi_note =
list(MIDI_NOTE_TO_NAME.keys())[list(MIDI_NOTE_TO_NAME.values()).index(note
)]
            pitch = midi_note + 12 * 5
            # Add note_on and note_off messages
            track.append(Message('note_on', note=pitch, velocity=64,
time=0))
            track.append(Message('note_off', note=pitch, velocity=64,
time=480))

    midi.save(output_path)

# Generated melody sequence
generated_sequence =
"FCaCCdCFgFdFgFRFFCaCCFddCaCagCCdCagFFRFCaCCdCdCagFFRFFggFgFaCCdCdCaggFFRF
FggFgFaCCdCdCagFFRFFggFgFaC"

# Output MIDI file path
output_path = "generated_melody.mid"
```

```
# Convert and save as MIDI
text_sequence_to_midi(generated_sequence, output_path)

print(f"Generated melody saved as {output_path}")
```