Notebook Overview: Tuning for HAN-GRU Model with Learning Rate=0.01

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
import re

import torch
from torch import nn
import torch.optim as optim
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import Dataset, DataLoader, random_split, RandomSampler, SequentialSampler

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer
from torch.utils.data import DataLoader, TensorDataset
from sklearn.decomposition import TruncatedSVD


from torchtext.data import get_tokenizer
from collections import Counter
from torchtext.vocab import Vocab, build_vocab_from_iterator
```

In [2]:
```
df = pd.read_csv("data/lyrics_cleaned.csv")
```

In [3]:
```
tokenizer = get_tokenizer('basic_english')
counter = Counter()
for line in tqdm(df['lyrics']):
    counter.update(tokenizer(line))
```

```
100%|██████████| 218162/218162 [00:23<00:00, 9461.18it/s]
```

In [4]:
```
# Create vocabulary using build_vocab_from_iterator
vocab = build_vocab_from_iterator([tokenizer(line) for line in df['lyrics']],
                                  specials=['<unk>', '<pad>'], min_freq=1)
```

In [5]:
```
label_encoder = LabelEncoder()
indexed_data = [torch.tensor([vocab[token] for token in tokenizer(line)])
                for line in df['lyrics']]

# Include padding for same shape size
max_seq_length = max(len(seq) for seq in indexed_data)
```

In [6]:
```
padded_data = pad_sequence(indexed_data, batch_first=True, padding_value=vocab['<pad>'])
indexed_labels = torch.tensor(label_encoder.fit_transform(df['genre']))
```

In [7]:
```
class LyricsDataset(Dataset):
    def __init__(self, lyrics, genre):
        self.lyrics = lyrics
        self.genre = genre

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

    def __getitem__(self, idx):
        return self.lyrics[idx], self.genre[idx]

dataset = LyricsDataset(padded_data, indexed_labels)

# Split into training and validation sets
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size

train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

print('{:>5,} training samples'.format(train_size))
print('{:>5,} validation samples'.format(val_size))
```

```
174,529 training samples
43,633 validation samples
```

In [8]:
```
batch_size=32
```

```
train_dataloader = DataLoader(
            train_dataset,
            sampler = RandomSampler(train_dataset),
            batch_size = batch_size
        )

validation_dataloader = DataLoader(
            val_dataset,
            sampler = SequentialSampler(val_dataset),
            batch_size = batch_size
        )
```

HAN Model

In [24]:
```
import torch
import torch.nn as nn

class WordAttention(nn.Module):
    def __init__(self, hidden_size):
        super(WordAttention, self).__init__()
        self.word_weights = nn.Linear(hidden_size, 1)

    def forward(self, rnn_output):
        word_scores = self.word_weights(rnn_output).squeeze(-1)
        word_scores = torch.softmax(word_scores, dim=-1)
        weighted_rnn_output = rnn_output * word_scores.unsqueeze(-1)
        sentence_embeddings = torch.sum(weighted_rnn_output, dim=1)
        return sentence_embeddings, word_scores

class SentenceAttention(nn.Module):
    def __init__(self, hidden_size):
        super(SentenceAttention, self).__init__()
        self.sentence_weights = nn.Linear(hidden_size, 1)

    def forward(self, rnn_output):
        sentence_scores = self.sentence_weights(rnn_output).squeeze(-1)
        sentence_scores = torch.softmax(sentence_scores, dim=-1)
        weighted_rnn_output = rnn_output * sentence_scores.unsqueeze(-1)
        document_embedding = torch.sum(weighted_rnn_output, dim=1)
        return document_embedding, sentence_scores
```

```
class HAN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, output_size):
        super(HAN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.word_rnn = nn.GRU(embedding_dim, hidden_size, batch_first=True, bidirectional=True)
        self.word_attention = WordAttention(hidden_size * 2)
        self.sentence_rnn = nn.GRU(hidden_size * 2, hidden_size, batch_first=True, bidirectional=True)
        self.sentence_attention = SentenceAttention(hidden_size * 2)
        self.fc = nn.Linear(hidden_size * 2, output_size)

    def forward(self, inputs):
        word_embedded = self.embedding(inputs)
        word_output, _ = self.word_rnn(word_embedded)
        sentence_embeddings, _ = self.word_attention(word_output)

        sentence_output, _ = self.sentence_rnn(word_output)
        document_embedding, _ = self.sentence_attention(sentence_output)

        output = self.fc(document_embedding)
        return output
```

In [27]:
```
num_epochs = 5
vocab_size = len(vocab)
embedding_dim = 512
hidden_size = 128
output_size = len(df['genre'].unique())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize your HAN model, optimizer, and criterion
model = HAN(vocab_size, embedding_dim, hidden_size, output_size).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.CrossEntropyLoss()
print(model)
```

```
HAN(
  (embedding): Embedding(75199, 512)
  (word_rnn): GRU(512, 128, batch_first=True, bidirectional=True)
  (word_attention): WordAttention(
    (word_weights): Linear(in_features=256, out_features=1, bias=True)
  )
  (sentence_rnn): GRU(256, 128, batch_first=True, bidirectional=True)
  (sentence_attention): SentenceAttention(
    (sentence_weights): Linear(in_features=256, out_features=1, bias=True)
  )
  (fc): Linear(in_features=256, out_features=11, bias=True)
)
```

In [28]:

```
for epoch in tqdm(range(num_epochs)):
    # Set model to training mode
    model.train()
    running_loss = 0.0
    num_correct = 0
    total = 0

    # Iterate over batches
    for inputs, labels in train_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)

        # Calculate loss
        loss = criterion(outputs, labels)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        _, predicted = torch.max(outputs, 1)  # Get the predicted class
        num_correct += (predicted == labels).sum().item()  # Accumulate correct predictions
        total += labels.size(0)

        running_loss += loss.item()
```

```
    # Calculate average training loss per epoch
    accuracy = 100 * num_correct / total  # Calculate accuracy for the epoch
    avg_train_loss = running_loss / len(train_dataloader)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {avg_train_loss:.4f}, Accuracy: {accuracy:.2f}%')
```

```
 20%|██        | 1/5 [04:17<17:08, 257.24s/it]
Epoch [1/5], Loss: 1.4360, Accuracy: 54.37%
 40%|████      | 2/5 [08:34<12:51, 257.21s/it]
Epoch [2/5], Loss: 1.1690, Accuracy: 62.01%
 60%|██████    | 3/5 [12:51<08:34, 257.30s/it]
Epoch [3/5], Loss: 0.8187, Accuracy: 73.91%
 80%|████████  | 4/5 [17:09<04:17, 257.62s/it]
Epoch [4/5], Loss: 0.4058, Accuracy: 87.61%
100%|██████████| 5/5 [21:28<00:00, 257.63s/it]
Epoch [5/5], Loss: 0.1618, Accuracy: 95.61%
```

In [29]:

```
# Validation loop (optional)
model.eval()
val_running_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for inputs, labels in validation_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        val_loss = criterion(outputs, labels)
        val_running_loss += val_loss.item()

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = correct / total
    avg_val_loss = val_running_loss / len(validation_dataloader)
    print(f'Validation Loss: {avg_val_loss:.4f}, Accuracy: {accuracy * 100:.2f}%')
```

```
Validation Loss: 2.4080, Accuracy: 51.22%
```

HAN-GRU Model

In [9]:
```
import torch
import torch.nn as nn

class AttLayer(nn.Module):
    def __init__(self, input_size, hidden_dim):
        super(AttLayer, self).__init__()
        self.hidden_dim = hidden_dim
        self.W = nn.Parameter(torch.randn(input_size, hidden_dim))
        self.bw = nn.Parameter(torch.zeros(hidden_dim))
        self.uw = nn.Parameter(torch.randn(hidden_dim))

    def forward(self, x):
        batch_size, num_words, hidden_size = x.size()
        x_reshaped = x.reshape(-1, hidden_size)

        ui = torch.tanh(torch.matmul(x_reshaped, self.W) + self.bw)
        intermed = torch.sum(self.uw * ui, dim=1)

        intermed = intermed.view(batch_size, num_words)
        weights = torch.softmax(intermed, dim=-1)
        weights = weights.unsqueeze(-1)

        weighted_input = x * weights
        return torch.sum(weighted_input, dim=1)

class HAN_GRU(nn.Module):
    def __init__(self, num_words, embedding_vector_length, hidden_size, attention_size, max_words_per_line
        super(HAN_GRU, self).__init__()

        self.word_embedding = nn.Embedding(num_words, embedding_vector_length)
        self.word_gru = nn.GRU(embedding_vector_length, hidden_size, batch_first=True, bidirectional=True)
        self.word_attention = AttLayer(hidden_size * 2, attention_size)

        self.sentence_gru = nn.GRU(hidden_size * 2, hidden_size, batch_first=True, bidirectional=True)
        self.sentence_attention = AttLayer(hidden_size * 2, attention_size)

        self.max_words_per_line = max_words_per_line
        self.max_num_lines = max_num_lines
```

```
        self.dropout = nn.Dropout(0.3)
        self.fc = nn.Linear(hidden_size * 2, output_size)

    def forward(self, inputs):
        word_embedded = self.word_embedding(inputs)

        # Word-level GRU
        word_output, _ = self.word_gru(word_embedded)
        word_attention_output = self.word_attention(word_output)

        # Reshape for sentence-level GRU
        batch_size = word_attention_output.size(0)
        sentence_input = word_attention_output.view(batch_size, -1, word_attention_output.size(-1))

        sentence_output, _ = self.sentence_gru(sentence_input)
        sentence_attention_output = self.sentence_attention(sentence_output)

        # Reshape for document-level output
        document_output = sentence_attention_output.view(batch_size, -1)
        output = self.fc(self.dropout(document_output))
        return output
```

In [10]:
```
# Calculate max words per line and max number of lines
max_words_per_line = df['lyrics'].apply(lambda x: len(x.split())).max()
max_num_lines = df['lyrics'].apply(lambda x: len(x.split('\n'))).max()
print(f"Max Words Per Line: {max_words_per_line}")
print(f"Max Number of Lines: {max_num_lines}")

attention_size = 100
hidden_size = 128
vocab_size = len(vocab)
embedding_dim = 512
output_size = len(df['genre'].unique())
```

```
Max Words Per Line: 6232
Max Number of Lines: 759
```

In [11]:
```
# Assuming you have your model, optimizer, criterion, and data loaders defined

num_epochs = 10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# Initialize your HAN-GRU model, optimizer, and criterion
model = HAN_GRU(vocab_size, embedding_dim, hidden_size, attention_size,
                max_words_per_line, max_num_lines, output_size).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

print(model)
```

```
HAN_GRU(
  (word_embedding): Embedding(245576, 512)
  (word_gru): GRU(512, 128, batch_first=True, bidirectional=True)
  (word_attention): AttLayer()
  (sentence_gru): GRU(256, 128, batch_first=True, bidirectional=True)
  (sentence_attention): AttLayer()
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=256, out_features=11, bias=True)
)
```

In [12]:
```
losses, accuracies = [], []

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for inputs, labels in tqdm(train_dataloader):
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        _, predicted = torch.max(outputs, 1)
        total_predictions += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()
```

```
    epoch_loss = running_loss / len(train_dataloader)
    epoch_accuracy = (correct_predictions / total_predictions) * 100
    losses.append(epoch_loss)
    accuracies.append(epoch_accuracy)

    print(f"Epoch [{epoch + 1}/{num_epochs}] Train Loss: {epoch_loss:.4f} Train Accuracy: {epoch_accuracy:
```

```
100%|██████████| 5455/5455 [10:52<00:00,  8.36it/s]
Epoch [1/10] Train Loss: 1.5594 Train Accuracy: 51.11%
100%|██████████| 5455/5455 [10:47<00:00,  8.42it/s]
Epoch [2/10] Train Loss: 1.4717 Train Accuracy: 53.57%
100%|██████████| 5455/5455 [10:48<00:00,  8.42it/s]
Epoch [3/10] Train Loss: 1.4163 Train Accuracy: 55.09%
100%|██████████| 5455/5455 [10:48<00:00,  8.41it/s]
Epoch [4/10] Train Loss: 1.3781 Train Accuracy: 56.12%
100%|██████████| 5455/5455 [10:48<00:00,  8.41it/s]
Epoch [5/10] Train Loss: 1.3427 Train Accuracy: 57.22%
100%|██████████| 5455/5455 [10:48<00:00,  8.41it/s]
Epoch [6/10] Train Loss: 1.3057 Train Accuracy: 58.25%
100%|██████████| 5455/5455 [10:47<00:00,  8.42it/s]
Epoch [7/10] Train Loss: 1.2691 Train Accuracy: 59.56%
100%|██████████| 5455/5455 [10:46<00:00,  8.43it/s]
Epoch [8/10] Train Loss: 1.2147 Train Accuracy: 61.29%
100%|██████████| 5455/5455 [10:45<00:00,  8.45it/s]
Epoch [9/10] Train Loss: 1.1511 Train Accuracy: 63.49%
100%|██████████| 5455/5455 [10:45<00:00,  8.45it/s]
Epoch [10/10] Train Loss: 1.0869 Train Accuracy: 65.59%
```

In [13]:

```
# Validation loop (optional)
model.eval()
val_running_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for inputs, labels in validation_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
```

```
        val_loss = criterion(outputs, labels)
        val_running_loss += val_loss.item()

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = correct / total
    avg_val_loss = val_running_loss / len(validation_dataloader)
    print(f'Validation Loss: {avg_val_loss:.4f}, Accuracy: {accuracy * 100:.2f}%')
```

```
Validation Loss: 1.2912, Accuracy: 59.64%
```

In [ ]:

```

```