Notebook Overview: Training Gated Recurrent Unit (GRU) Models

This notebook documents the training process of two models utilizing the Gated Recurrent Unit (GRU) architecture. Due to extended training durations, the initial 25,000 rows/lyrics from the dataset were employed.

The models under development are as follows:

- Bidirectional GRU Network
- Bidirectional GRU Network with a Dropout Layer

In [2]:
```
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 [12]:
```
df = pd.read_csv("data/lyrics_cleaned.csv")
```

In [13]:
```
df = df.sample(25000)
```

In [14]:
```
tokenizer = get_tokenizer('basic_english')
counter = Counter()
for line in tqdm(df['lyrics']):
    counter.update(tokenizer(line))
```

```
100%|██████████| 25000/25000 [00:02<00:00, 9362.72it/s]
```

In [15]:
```
# Create vocabulary using build_vocab_from_iterator
vocab = build_vocab_from_iterator([tokenizer(line) for line in df['lyrics'][:25000]],
                                  specials=['<unk>', '<pad>'], min_freq=1)
```

In [16]:
```
label_encoder = LabelEncoder()
indexed_data = [torch.tensor([vocab[token] for token in tokenizer(line)])
                for line in df['lyrics'][:25000]]

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

In [17]:
```
padded_data = pad_sequence(indexed_data, batch_first=True, padding_value=vocab['<pad>'])
indexed_labels = torch.tensor(label_encoder.fit_transform(df['genre'][:25000]))
```

In [18]:
```
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))
```

```
20,000 training samples
5,000 validation samples
```

In [19]:
```
batch_size=24

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
        )
```

Bidirectional GRU

In [20]:
```
class BidirectionalGRU(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_size):
        super(BidirectionalGRU, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru = nn.GRU(embedding_dim, hidden_dim, batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_size)  # Output size doubled due to bidirectionality

    def forward(self, x):
        embedded = self.embedding(x)
        gru_out, _ = self.gru(embedded)
        # Concatenate the last hidden state from both directions
        combined = torch.cat((gru_out[:, -1, :hidden_dim], gru_out[:, 0, hidden_dim:]), dim=1)
        output = self.fc(combined)
        return output
```

In [22]:
```
vocab_size = len(vocab)
embedding_dim = 512
hidden_dim = 128
output_size = len(df['genre'].unique())

# Initialize the model
model = BidirectionalGRU(vocab_size, embedding_dim, hidden_dim, output_size)
print(model)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 5  # Number of epochs
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(device)
```

```
BidirectionalGRU(
  (embedding): Embedding(72516, 512)
  (gru): GRU(512, 128, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=256, out_features=11, bias=True)
)
cuda
```

In [23]:
```
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 [00:32<02:11, 32.95s/it]
Epoch [1/5], Loss: 1.6804, Accuracy: 46.56%
 40%|████      | 2/5 [01:05<01:37, 32.62s/it]
Epoch [2/5], Loss: 1.4542, Accuracy: 53.82%
 60%|██████    | 3/5 [01:37<01:05, 32.50s/it]
Epoch [3/5], Loss: 1.2246, Accuracy: 60.09%
 80%|████████  | 4/5 [02:10<00:32, 32.45s/it]
Epoch [4/5], Loss: 0.9845, Accuracy: 68.10%
100%|██████████| 5/5 [02:42<00:00, 32.48s/it]
Epoch [5/5], Loss: 0.7146, Accuracy: 77.11%
```

In [24]:

```
# 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: 1.7380, Accuracy: 50.40%
```

Bidirectional GRU with Dropout Layer

Observing indications of overfitting in our previous bidirectional GRU architecture, evident through rapid loss reduction and notably high accuracy, we've introduced a dropout layer as a strategy to address this issue.

In [25]:

```
class BidirectionalGRU(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_size, dropout=0.2):
        super(BidirectionalGRU, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(embedding_dim, hidden_dim, batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_size)

    def forward(self, x):
        embedded = self.embedding(x)
        embedded = self.dropout(embedded)
        gru_out, _ = self.gru(embedded)
        combined = torch.cat((gru_out[:, -1, :hidden_dim], gru_out[:, 0, hidden_dim:]), dim=1)
        output = self.fc(combined)
        return output
```

In [26]:

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

# Initialize the model
model = BidirectionalGRU(vocab_size, embedding_dim, hidden_dim, output_size)
print(model)
```

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
num_epochs = 5  # Number of epochs
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(device)
```

```
BidirectionalGRU(
  (embedding): Embedding(72516, 512)
  (dropout): Dropout(p=0.2, inplace=False)
  (gru): GRU(512, 128, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=256, out_features=11, bias=True)
)
cuda
```

In [27]:

```python
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 [00:32<02:10, 32.56s/it]
Epoch [1/5], Loss: 1.6884, Accuracy: 46.70%
 40%|████      | 2/5 [01:04<01:37, 32.47s/it]
Epoch [2/5], Loss: 1.4560, Accuracy: 53.45%
 60%|██████    | 3/5 [01:37<01:04, 32.44s/it]
Epoch [3/5], Loss: 1.2582, Accuracy: 59.53%
 80%|████████  | 4/5 [02:09<00:32, 32.45s/it]
Epoch [4/5], Loss: 1.0623, Accuracy: 66.14%
100%|██████████| 5/5 [02:42<00:00, 32.46s/it]
Epoch [5/5], Loss: 0.8841, Accuracy: 71.58%
```

In [28]:

```
# 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: 1.5735, Accuracy: 54.66%
```

In [ ]:

```
```