CS 405: Deep Learning | Lab

Experiment 3: Implement Music Generation using Recurrent Neural Networks in PyTorch

Student Name: Nabeel Shan
Registration No: 468752

Objective

To develop a Recurrent Neural Network (RNN) using the **Long Short-Term Memory (LSTM)** architecture in PyTorch. The goal is to train the model on musical compositions in **ABC notation** and generate original, melodically coherent music.

```
# !pip install comet_ml
```

```
# !pip install comet_ml mitdeeplearning --quiet
# !apt-get install abcmidi timidity -y
```

1. Environment Setup & Data Preprocessing

We utilize the **MIT Deep Learning (mdl)** library to handle the specific music dataset. The data consists of thousands of Irish folk songs in ABC notation.

Preprocessing Steps:

1. **Load Data:** Import the catalog of songs.
2. **Vectorization:** Create a mapping from unique characters to integers (`char2idx`) and vice versa (`idx2char`).
3. **Batching:** The data is joined into a single stream and sliced into sequences (e.g., length 100) to feed the LSTM.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import os
import time
from tqdm import tqdm
from scipy.io.wavfile import write
from IPython import display as ipythondisplay
import mitdeeplearning as mdl
```

```
assert torch.cuda.is_available(), "Enable GPU from Runtime settings"
```

```
songs = mdl.lab1.load_training_data()

example_song = songs[6]
print("Example song:\n")
print(example_song)

mdl.lab1.play_song(example_song)
```

Prepare Training Data

```
songs_joined = "\n\n".join(songs)

vocab = sorted(set(songs_joined))
char2idx = {u:i for i,u in enumerate(vocab)}
idx2char = np.array(vocab)

print("Vocabulary size:", len(vocab))
```

```
Vocabulary size: 83
```

```
def vectorize_string(string):
    return np.array([char2idx[c] for c in string], dtype=np.int32)

vectorized_songs = vectorize_string(songs_joined)
```

```
Start coding or generate with AI.
```

2. The Recurrent Neural Network (LSTM)

We implement a custom `LSTMModel` class subclassing `nn.Module`.

Architecture:

- **Embedding Layer:** Converts integer indices into dense vectors.
- **LSTM Layer:** Processes the sequence data, maintaining a hidden state (h_t) and cell state (c_t) to capture long-term dependencies in the music structure.
- **Dense Layer (FC):** Maps the LSTM output back to the vocabulary size to predict the next character.

```
class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, num_layers=2):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(
            embedding_dim,
            hidden_size,
            num_layers=num_layers,
            batch_first=True
        )
        self.fc = nn.Linear(hidden_size, vocab_size)

    def init_hidden(self, batch_size, device):
        return (
            torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
            torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
        )

    def forward(self, x, state=None, return_state=False):
        x = self.embedding(x)
        if state is None:
            state = self.init_hidden(x.size(0), x.device)
        out, state = self.lstm(x, state)
        out = self.fc(out)
        return (out, state) if return_state else out
```

Training Utilities

```
def get_batch(data, seq_length, batch_size):
    n = data.shape[0] - 1
    idx = np.random.choice(n - seq_length, batch_size)

    x = np.array([data[i:i+seq_length] for i in idx], dtype=np.int64)
    y = np.array([data[i+1:i+seq_length+1] for i in idx], dtype=np.int64)

    return (
        torch.tensor(x, dtype=torch.long),
        torch.tensor(y, dtype=torch.long)
    )
```

```
def compute_loss(labels, logits):
    labels = labels.view(-1)
    logits = logits.view(-1, logits.shape[-1])
    return nn.CrossEntropyLoss()(logits, labels)
```

3. Training the Model

We train the model to minimize the **Cross Entropy Loss** between the predicted character and the actual next character in the sequence.

Hyperparameters:

- **Batch Size:** 8
- **Sequence Length:** 100 characters
- **Hidden Size:** 1024 (Capacity of the LSTM)
- **Embedding Dim:** 256
- **Optimizer:** Adam (Learning Rate: 0.005)

```
params = {
    "num_training_iterations": 3000,
    "batch_size": 8,
    "seq_length": 100,
    "learning_rate": 5e-3,
    "embedding_dim": 256,
    "hidden_size": 1024
}

device = torch.device("cuda")

model = LSTMModel(
    len(vocab),
    params["embedding_dim"],
    params["hidden_size"]
).to(device)

optimizer = optim.Adam(model.parameters(), lr=params["learning_rate"])
```

```
Start coding or generate with AI.
```

3.1 Training Loop

The model iterates through the dataset for 3000 steps. We use `tqdm` to visualize progress. Checkpoints are saved periodically to allow for model recovery or inference later.

```
checkpoint_dir = "./training_checkpoints"
os.makedirs(checkpoint_dir, exist_ok=True)

for step in tqdm(range(params["num_training_iterations"])):
    x, y = get_batch(vectorized_songs, params["seq_length"], params["batch_size"])
    x, y = x.to(device), y.to(device)

    model.train()
    optimizer.zero_grad()

    y_hat, _ = model(x, return_state=True)
    loss = compute_loss(y, y_hat)

    loss.backward()
    optimizer.step()

    if step % 500 == 0:
        torch.save(model.state_dict(), f"{checkpoint_dir}/my_ckpt")
```

```
100%|██████████| 3000/3000 [03:02<00:00, 16.44it/s]
```

```
Start coding or generate with AI.
```

4. Music Generation

Now that the model has learned the statistical patterns of ABC notation (including headers like `K:` for key and `M:` for meter), we can generate new songs.

Generation Process:

1. **Initialization:** Seed the model with a start character (e.g., "X").
2. **Sampling:** At each step, sample the next character from the probability distribution output by the model (using `torch.multinomial`).
3. **Decoding:** Convert the integer sequence back to ABC notation.
4. **Audio Conversion:** Use `abcmidi` and `timidity` to render the ABC text into a `.wav` file.

```
def generate_text(model, start_string="X", length=3000):
    model.eval()

    input_idx = torch.tensor([char2idx[c] for c in start_string]).unsqueeze(0).to(device)
    state = model.init_hidden(1, device)

    generated = []

    for _ in range(length):
        logits, state = model(input_idx, state, return_state=True)
        probs = torch.softmax(logits[:, -1, :], dim=-1)
        next_id = torch.multinomial(probs, num_samples=1)
        generated.append(idx2char[next_id.item()])
        input_idx = next_id

    return start_string + "".join(generated)
```

```
model.load_state_dict(torch.load("./training_checkpoints/my_ckpt"))

generated_text = generate_text(model)
songs = mdl.lab1.extract_song_snippet(generated_text)

for i, song in enumerate(songs):
    print(f"Generated Song {i}")
    mdl.lab1.play_song(song)
```

```
Found 11 songs in text
Generated Song 0
Generated Song 1
Generated Song 2
Generated Song 3
Generated Song 4
Generated Song 5
Generated Song 6
Generated Song 7
Generated Song 8
Generated Song 9
Generated Song 10
```

```
generated_songs = mdl.lab1.extract_song_snippet(generated_text)

print(f"Number of generated songs: {len(generated_songs)}")
```

```
Found 11 songs in text
Number of generated songs: 11
```

4.1 Audio Rendering & Playback

We extract valid song snippets from the generated text and convert them to audio waveforms for playback.

```
for i, song in enumerate(generated_songs):
    print(f"\n🎵 Generated Song {i}\n")
    print(song)   # show ABC notation (optional)
    mdl.lab1.play_song(song)
```

```
🎵 Generated Song 0

X:69
T:Tevers' Pught
Z: id:dc-hornpipe-40
M:C|
L:1/8
K:G Major
F|g2d Bcd|edc Bc|d6|[1 c>B c|[1 F2 AB/c/|[1 B/c dc|BA F>E|F/D/F G|D2:|!
```

```
Generated Song 1


X:44
T:Mabin Hilltor
Z: id:dc-jig-340
M:6/8
L:1/8
K:G Major
D|G3 GAB|ded dBA|Bee edB|dBA G2B|!
[1 GED G2A|BAB d2A|BAB def|edB ABA|GEE E2:|!
[2 GA|BG ED|EDE GAB|cBA B<d|efe dBA|G2A BA2|]!
e|ede dBA|BdB dBA|BGF G2F|GEE E2|]!

Generated Song 2

X:44
T:Mabin Hilltor
Z: id:dc-jig-340
M:6/8
L:1/8
K:G Major
D|G3 GAB|ded dBA|Bee edB|dBA G2B|!
[1 GED G2A|BAB d2A|BAB def|edB ABA|GEE E2:|!
[2 GA|BG ED|EDE GAB|cBA B<d|efe dBA|G2A BA2|]!
e|ede dBA|BdB dBA|BGF G2F|GEE E2|]!

Generated Song 3

X:5
T:Forty
Z: id:dc-polka-6
M:2/4
L:1/8
K:D Major
B,|A,2D2 FEDC|A,DFD B,DFD|cFdF =FEDE|F2ED EFG:|!
FDd,D B,DFD|CDEG FDD2|B,DFD CEE2|FEDF GFGA|!
Bdd^c d2AF|DFFd AFFA|BAFA BGGF|AddA BGE:|!
F|Dddc BAGF|DFAF BFAB|Beed cAAc|BdAG FDD:|!

Generated Song 4

X:142
T:Sailor on Tinker
Z: id:dc-jig-166
M:6/8
```

```
for i, song in enumerate(generated_songs):
    waveform = mdl.lab1.play_song(song)

    if waveform is not None:
        audio_data = np.frombuffer(waveform.data, dtype=np.int16)
        filename = f"generated_song_{i}.wav"
        write(filename, 88200, audio_data)
        print(f"Saved: {filename}")
```

```
Saved: generated_song_0.wav
Saved: generated_song_1.wav
Saved: generated_song_2.wav
Saved: generated_song_3.wav
Saved: generated_song_4.wav
Saved: generated_song_5.wav
Saved: generated_song_6.wav
Saved: generated_song_7.wav
Saved: generated_song_8.wav
Saved: generated_song_9.wav
Saved: generated_song_10.wav
```

```
for i, song in enumerate(generated_songs):
    mdl.lab1.play_song(song)
    break
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
# !apt-get install abcmidi timidity -y
```

```
playable_songs = []

for song in generated_songs:
    if "K:" in song and "|" in song:
        playable_songs.append(song)

print(f"Playable songs: {len(playable_songs)}")
```

```
Playable songs: 11
```

```
from IPython.display import Audio, display

for i in range(len(generated_songs)):
    display(Audio(f"generated_song_{i}.wav"))
```

0:00 / 0:16

0:00 / 0:34

0:00 / 0:34

0:00 / 1:05

```
Buffered data was truncated after reaching the output size limit.
```

```
Start coding or generate with AI.
```