Name: **B. SAI CHARAN**

Roll No: **2203A51L72**

Batch No: **21CSBTB12**

ASSIGNMENT – 6

Question:

Implement and compare Long sort term memory and Gated recurrent Unit models for text generation.

Answer:

To implement and compare Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models for text generation, here's an outline of the process in Python, using PyTorch. This example uses simple text data for demonstration, where each model generates text based on a sequence of characters.

Implementation Steps

1. **Data Preparation**: Preprocess the text data, converting characters to indices for the model.
2. **Model Definition**: Define two models, one using LSTM and another using GRU.
3. **Training and Comparison**: Train both models and compare their performance.

Explanation:

- **Data Preparation**: The text data is tokenized into characters and encoded as integers.
- **Model Definition**: Both models have an LSTM and a GRU variant, each with two layers and a fully connected layer to predict the next character.
- **Training**: Both models are trained independently, and their performance (loss) is tracked for comparison.
- **Text Generation**: Text is generated by feeding an initial character sequence and letting the model predict the next character iteratively.

Code Implementation:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random

# Sample text data
text = "long short term memory and gated recurrent units are used in text generation"
chars = sorted(set(text))
char_to_idx = {ch: i for i, ch in enumerate(chars)}
idx_to_char = {i: ch for i, ch in enumerate(chars)}

# Convert text to integers
data = [char_to_idx[ch] for ch in text]
input_size = len(chars)

# Hyperparameters
hidden_size = 128
num_layers = 2
seq_length = 20
batch_size = 1
num_epochs = 100
learning_rate = 0.002

# Model definitions
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, h):
        out, h = self.lstm(x, h)
        out = self.fc(out[:, -1, :])  # Output for last sequence element
        return out, h

class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
```

Text Data and Character Encoding

We start by defining our text data, which we'll use to train the model for text generation. Each character in the text is assigned a unique integer index, which allows the model to work with numeric data.

chars: Unique characters in the text, e.g., [' ', 'a', 'd', 'e', ...].

char_to_idx and **idx_to_char**: Dictionaries that map each character to a unique integer index and vice versa.

data: The text data converted to integers. This lets us work with indexed inputs and targets.

```
    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.fc(out[:, -1, :])
        return out, h

# Initialize models
lstm_model = LSTMModel(input_size, hidden_size, input_size, num_layers)
gru_model = GRUModel(input_size, hidden_size, input_size, num_layers)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
lstm_optimizer = optim.Adam(lstm_model.parameters(), lr=learning_rate)
gru_optimizer = optim.Adam(gru_model.parameters(), lr=learning_rate)

# Training function
def train(model, optimizer):
    model.train()
    h = None
    total_loss = 0
    for i in range(0, len(data) - seq_length, seq_length):
        inputs = torch.eye(input_size)[data[i:i+seq_length]].unsqueeze(0)
        targets = torch.tensor(data[i+1:i+seq_length+1])

        # Forward pass
        output, h = model(inputs, h)
        h = tuple([state.detach() for state in h]) if isinstance(h, tuple) else h.detach()

        # Calculate loss and perform backward pass
        loss = criterion(output, targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
    return total_loss / (len(data) // seq_length)
```

Model Definitions

Here we define two models: LSTMModel and GRUModel. Each model takes input characters and produces output predictions for the next character.

LSTMModel: This model contains an LSTM layer and a fully connected layer for the final prediction.

h: Initial hidden state for the LSTM, which helps in remembering previous sequence information.

fc layer: Fully connected layer that maps LSTM's hidden state output to the final character prediction.

```python
# Generate text function
def generate_text(model, start_text="text ", length=50):
    model.eval()
    h = None
    chars_input = [char_to_idx[ch] for ch in start_text]
    inputs = torch.eye(input_size)[chars_input].unsqueeze(0)
    result = start_text

    for _ in range(length):
        output, h = model(inputs, h)
        predicted_idx = torch.argmax(output).item()
        result += idx_to_char[predicted_idx]

        # Prepare next input
        inputs = torch.eye(input_size)[[predicted_idx]].unsqueeze(0)

    return result

# Training loop and comparison
for epoch in range(num_epochs):
    lstm_loss = train(lstm_model, lstm_optimizer)
    gru_loss = train(gru_model, gru_optimizer)
    if epoch % 10 == 0:
        print(f'Epoch [{epoch}/{num_epochs}], LSTM Loss: {lstm_loss:.4f}, GRU Loss: {gru_loss:.4f}')

# Generate text with both models
print("LSTM Generated Text:\n", generate_text(lstm_model, "text "))
print("GRU Generated Text:\n", generate_text(gru_model, "text "))
```

After training, text is generated using both models to observe any qualitative differences in the generated sequences. Both LSTM and GRU models are capable of capturing patterns in text, but their architecture differences can affect generation quality and efficiency.

This structure provides a good basis for understanding LSTM and GRU behaviors in text generation tasks.