CSET419 – Introduction to Generative AI

Lab – 4: Text Generation Models

Submitted By: Deepanshu
Date: 2026-02-04

Objective

The objective of this lab is to design and implement a simple text generation model that can learn patterns from a given text corpus and generate new, meaningful text sequences.

Learning Outcomes

- Understand the basics of text generation
- Preprocess textual data for neural networks
- Implement a sequence-based neural network model (RNN/LSTM/GRU)
- Implement a Transformer-based model
- Generate new text using a trained model

Component-I: RNN / LSTM Based Text Generation

This section demonstrates text generation using Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), and Gated Recurrent Units (GRU).

1. Dataset & Preprocessing

A custom text corpus was used. Tokenization implies character-level mapping.

Dataset Snippet:

```
artificial intelligence is transforming modern society.
it is used in healthcare finance education and transportation.
machine learning allows systems to improve automatically with experience.
data plays a critical role in training intelligent systems.
large datasets help models learn complex patter
...
```

2. Simple RNN Implementation

Implementation of a vanilla RNN using `nn.RNN`.

Code (rnn_gen.py):

```
import torch
import torch.nn as nn
import torch.optim as optim
import os
import utils

MODEL_FILE = 'rnn_model.pth'
SEQ_LENGTH = 40
HIDDEN_SIZE = 128
NUM_LAYERS = 1

class TextGeneratorRNN(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers):
        super(TextGeneratorRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        embed = self.embedding(x)
        out, hidden = self.rnn(embed, hidden)
        out = self.fc(out[:, -1, :])
        return out, hidden

def train_model():
    text, dataX, dataY, char_to_int, _, vocab_size = utils.load_data(SEQ_LENGTH)
    if text is None: return

    X = torch.tensor(dataX, dtype=torch.long)
    y = torch.tensor(dataY, dtype=torch.long)
    dataset = torch.utils.data.TensorDataset(X, y)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if torch.backends.mps.is_available(): device = torch.device('mps')

    model = TextGeneratorRNN(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.005)

    print("Starting Training (Vanilla RNN)...")
    for epoch in range(100):
        model.train()
        total_loss = 0
        for batch_x, batch_y in dataloader:
```

```
                batch_x, batch_y = batch_x.to(device), batch_y.to(device)
                optimizer.zero_grad()
                hidden = None
                output, _ = model(batch_x, hidden)
                loss = criterion(output, batch_y)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()

            if (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}/100, Loss: {total_loss/len(dataloader):.4f}")

        torch.save(model.state_dict(), MODEL_FILE)
        print(f"RNN model saved to {MODEL_FILE}")

    def generate_text_model(start_str, length=200):
        if not os.path.exists(MODEL_FILE):
            print("Model not found. Please train first.")
            return ""

        _, _, _, char_to_int, int_to_char, vocab_size = utils.load_data(SEQ_LENGTH)

        device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        if torch.backends.mps.is_available(): device = torch.device('mps')

        model = TextGeneratorRNN(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)
        model.load_state_dict(torch.load(MODEL_FILE, map_location=device))
        model.eval()

        current_string = start_str.lower()
        generated_text = start_str

        with torch.no_grad():
            for _ in range(length):
                if len(current_string) >= SEQ_LENGTH: input_seq = current_string[-SEQ_LENGTH:]
                else: input_seq = current_string

                input_indices = [char_to_int.get(c, 0) for c in input_seq]
                input_tensor = torch.tensor(input_indices, dtype=torch.long).unsqueeze(0).to(device)

                output, _ = model(input_tensor, None)
                probs = torch.softmax(output, dim=1)
                next_char_idx = torch.multinomial(probs, 1).item()

                next_char = int_to_char[next_char_idx]
                generated_text += next_char
                current_string += next_char

        return generated_text

    if __name__ == "__main__":
        train_model()
```

Generated Output (RNN):

```
artificial intelligent systems.
large datasets help models are important concerns.
researchers continue to improve ai systems should be meaning.
automatically with experiences.

ethical consid on languages trai
```

3. LSTM & GRU Implementation

LSTM and GRU architectures address the vanishing gradient problem in simple RNNs.

Code (lstm_gru_gen.py):

```
import torch
import torch.nn as nn
import torch.optim as optim
import os
import utils

SEQ_LENGTH = 40
HIDDEN_SIZE = 256
NUM_LAYERS = 2

class TextGeneratorLSTM(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers):
        super(TextGeneratorLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        embed = self.embedding(x)
        out, hidden = self.lstm(embed, hidden)
        out = self.fc(out[:, -1, :])
        return out, hidden

class TextGeneratorGRU(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers):
        super(TextGeneratorGRU, self).__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden):
        embed = self.embedding(x)
        out, hidden = self.gru(embed, hidden)
        out = self.fc(out[:, -1, :])
        return out, hidden

def train_model(model_type='LSTM'):
    text, dataX, dataY, _, _, vocab_size = utils.load_data(SEQ_LENGTH)
    if text is None: return

    X = torch.tensor(dataX, dtype=torch.long)
    y = torch.tensor(dataY, dtype=torch.long)
    dataset = torch.utils.data.TensorDataset(X, y)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if torch.backends.mps.is_available(): device = torch.device('mps')

    if model_type == 'LSTM':
        model = TextGeneratorLSTM(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)
        model_file = 'lstm_model.pth'
    else:
        model = TextGeneratorGRU(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)
        model_file = 'gru_model.pth'

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.002)

    print(f"Starting Training ({model_type})...")
    for epoch in range(100):
        model.train()
        total_loss = 0
        for batch_x, batch_y in dataloader:
```

```
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            optimizer.zero_grad()
            hidden = None
            output, _ = model(batch_x, hidden)
            loss = criterion(output, batch_y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}/100, Loss: {total_loss/len(dataloader):.4f}")

    torch.save(model.state_dict(), model_file)
    print(f"{model_type} model saved to {model_file}")

def generate_text_model(start_str, model_type='LSTM', length=200):
    model_file = 'lstm_model.pth' if model_type == 'LSTM' else 'gru_model.pth'

    if not os.path.exists(model_file):
        print(f"Model {model_file} not found. Please train first.")
        return ""

    _, _, _, char_to_int, int_to_char, vocab_size = utils.load_data(SEQ_LENGTH)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if torch.backends.mps.is_available(): device = torch.device('mps')

    if model_type == 'LSTM':
        model = TextGeneratorLSTM(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)
    else:
        model = TextGeneratorGRU(vocab_size, HIDDEN_SIZE, NUM_LAYERS).to(device)

    model.load_state_dict(torch.load(model_file, map_location=device))
    model.eval()

    current_string = start_str.lower()
    generated_text = start_str

    with torch.no_grad():
        for _ in range(length):
            if len(current_string) >= SEQ_LENGTH: input_seq = current_string[-SEQ_LENGTH:]
            else: input_seq = current_string

            input_indices = [char_to_int.get(c, 0) for c in input_seq]
            input_tensor = torch.tensor(input_indices, dtype=torch.long).unsqueeze(0).to(device)

            output, _ = model(input_tensor, None)
            probs = torch.softmax(output, dim=1)
            next_char_idx = torch.multinomial(probs, 1).item()

            next_char = int_to_char[next_char_idx]
            generated_text += next_char
            current_string += next_char

    return generated_text

if __name__ == "__main__":
    train_model('LSTM')
```

Generated Output (LSTM):

```
artificial intelligence.
fairness transparency and accountability must be ensured.
ai systems should be designed responsibly.
data privacy and security are major concerns.
researchers continue to improve ai saf
```

Generated Output (GRU):

```
artificial intelligence.
```

fairness transparency and accountability must be ensured.
ai systems should be designed responsibly.
data privacy and security are major concerns.
researchers continue to improve ai saf

Component-II: Transformer Based Text Generation

Transformer models rely on self-attention mechanisms to process sequential data in parallel, offering significant performance improvements over RNNs for many tasks.

Implementation Details

Uses `nn.TransformerEncoder` with Positional Encoding to retain sequence order information.

Code (transformer_gen.py):

```
import torch
import torch.nn as nn
import torch.optim as optim
import math
import os
import utils

MODEL_FILE = 'transformer_model.pth'
SEQ_LENGTH = 40
D_MODEL = 128
NHEAD = 4
NUM_LAYERS = 2

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:, :x.size(1), :]

class TextGeneratorTransformer(nn.Module):
    def __init__(self, vocab_size, d_model, nhead, num_layers):
        super(TextGeneratorTransformer, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model)
        encoder_layers = nn.TransformerEncoderLayer(d_model, nhead, batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers)
        self.fc = nn.Linear(d_model, vocab_size)
        self.d_model = d_model

    def forward(self, x):
        embed = self.embedding(x) * math.sqrt(self.d_model)
        src = self.pos_encoder(embed)
        output = self.transformer_encoder(src)
        out = self.fc(output[:, -1, :])
        return out

def train_model():
    text, dataX, dataY, _, _, vocab_size = utils.load_data(SEQ_LENGTH)
    if text is None: return

    X = torch.tensor(dataX, dtype=torch.long)
    y = torch.tensor(dataY, dtype=torch.long)
    dataset = torch.utils.data.TensorDataset(X, y)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if torch.backends.mps.is_available(): device = torch.device('mps')

    model = TextGeneratorTransformer(vocab_size, D_MODEL, NHEAD, NUM_LAYERS).to(device)
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0005)

    print("Starting Training (Transformer)...")
    for epoch in range(100):
        model.train()
        total_loss = 0
        for batch_x, batch_y in dataloader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            optimizer.zero_grad()
            output = model(batch_x)
            loss = criterion(output, batch_y)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}/100, Loss: {total_loss/len(dataloader):.4f}")

    torch.save(model.state_dict(), MODEL_FILE)
    print(f"Transformer model saved to {MODEL_FILE}")

def generate_text_model(start_str, length=200):
    if not os.path.exists(MODEL_FILE):
        print("Model not found. Please train first.")
        return ""

    _, _, _, char_to_int, int_to_char, vocab_size = utils.load_data(SEQ_LENGTH)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    if torch.backends.mps.is_available(): device = torch.device('mps')

    model = TextGeneratorTransformer(vocab_size, D_MODEL, NHEAD, NUM_LAYERS).to(device)
    model.load_state_dict(torch.load(MODEL_FILE, map_location=device))
    model.eval()

    current_string = start_str.lower()
    generated_text = start_str

    with torch.no_grad():
        for _ in range(length):
            if len(current_string) >= SEQ_LENGTH: input_seq = current_string[-SEQ_LENGTH:]
            else: input_seq = current_string

            input_indices = [char_to_int.get(c, 0) for c in input_seq]
            input_tensor = torch.tensor(input_indices, dtype=torch.long).unsqueeze(0).to(device)

            output = model(input_tensor)

            probs = torch.softmax(output, dim=1)
            next_char_idx = torch.multinomial(probs, 1).item()

            next_char = int_to_char[next_char_idx]
            generated_text += next_char
            current_string += next_char

    return generated_text

if __name__ == "__main__":
    train_model()
```

Generated Output (Transformer):

```
artificial trrnt sp.
t utfove an ange agen as agelatines cy leleritemodetems parntes.
isthatanthint arngrexlli mperstalalarnsterncct mpattate matatroncles.
tentowscy sso sto she to tonitysy mon bon alon .
a qua
```

Conclusion

In this lab, we successfully implemented and compared N-gram, RNN, LSTM, GRU, and Transformer models for character-level text generation. While simple RNNs can generate text, LSTM and GRU models show better capability in capturing longer dependencies. Transformer models, although more complex, provide a powerful architecture for sequence tasks.