EXPERIMENT 4:- DOCUMENTATION

Delhi Technological University
Shahbad Daulatpur Village, Rohini, New Delhi, Delhi, 110042

(Department of Computer science and technology)
(Deep learning: CO328 E5 Sec1 G1)

SUBMITTED BY:-
Aryan Gahlaut

SUBMITTED TO:-
Dr Anil Singh Parihar

2K22/CO/102

6th semester (3rd year)

EXPERIMENT 4

Aim: The aim of this experiment is to explore text generation using Recurrent Neural Networks (RNNs) and understand the impact of different word representations:
1. One-Hot Encoding
2. Trainable Word Embeddings

Train an RNN model on a dataset of 100 poems and compare the performance of both encoding techniques.

Dataset used: Kaggle's "100 Poem Dataset" includes 100 poems and provides a mix of different poetic forms, themes, and word choice. The dataset is well suited for natural language processing (NLP) applications like sentiment analysis, text classification, and poem generation.There are different poems available in the dataset of varying structures and lengths penned by renowned poets and anonymous writers.It gives a comprehensive idea of the history of poetry with detailed linguistic structures that can be analyzed and utilized for machine learning. Data scientists, researchers, and machine learning experts can train models on this dataset that comprehend poetic language as well as the subtleties surrounding it. It is particularly useful for those who are interested in learning about the intersection of artificial intelligence and arts.

Platform used: Google colab
It is an end-to-end cloud-based solution for writing and executing the Python code from a user's web browser called Google Colab, or also known as Google Colaboratory. This feature allows the user to work interactively in the same environment as many popular libraries: TensorFlow, PyTorch, and NumPy, especially in demand applications such as machine learning, data analysis, and research.
One of the advantages of Google Colab is access to free computing resources: GPUs and TPUs, that may accelerate the computation significantly and is especially needed for training big models , provided the access to these GPUs and TPUs are limited for a limited amount of time.
The platform integrates Google Drive and supports saving and easy sharing of the work. It also supports Jupyter notebooks, which provide a rich interface that allows users to combine code, visualizations, and text in a single document, making it an excellent tool for collaboration and learning.

Libraries and framework used:

1) Torch: Torch is a deep learning framework that has been developed to build and train neural networks. It supports tools for tensor computation, automatic differentiation, and GPU acceleration**.**

2) torch.nn: A submodule of PyTorch that contains all the building blocks for constructing neural networks, from layers to loss functions and optimization algorithm.

3) pandas: is an efficient data manipulation library primarily used in handling structured data that includes tables. Data cleaning, analysis, and transformation are some of the tools that it provides.

4) torch.utils.data.TensorDataset: A PyTorch utility for creating datasets from tensors, which then can be used with DataLoader for batching and shuffling during training.

5) torch.utils.data.DataLoader: PyTorch is a tool for loading batches of the data during training where the data optionally gets shuffled.

6) torch.nn.functional: PyTorch module that provides functional versions of many operations, including activation functions and loss functions, used in the definition of models.

7) random: It will help us to select words randomly with highest possibilities of being next in sequence during poem generation.

Implementation1: One hot encoding approach

Importing necessary libraries and framework and pre processing

```
import torch
import torch.nn as nn
import pandas as pd
import random
import string
from torch.utils.data import Dataset, DataLoader


device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# gpu availabi;ity
print(f"Using device: {device}")
```

```
df = pd.read_csv("/content/poems-100.csv")                                    # loading poem
poems = df['text'].tolist()


def clean_text(text):                                                                  #
cleaning  text
    text = text.lower()  # Lowercase the text
    text = text.translate(str.maketrans('', '', string.punctuation))  # Remove
punctuation
    return text


poems = [clean_text(poem) for poem in poems]                                          #
tokenization
all_words = [word for poem in poems for word in poem.split()]
vocab = sorted(list(set(all_words)))
word_to_idx = {word: i for i, word in enumerate(vocab)}
idx_to_word = {i: word for i, word in enumerate(vocab)}
vocab_size = len(vocab)


def one_hot_encode(word_idx, vocab_size):                                               #
one hot encoding function
    one_hot = torch.zeros(vocab_size)
    one_hot[word_idx] = 1
    return one_hot


sequences = []
for poem in poems:                                                                   ##
Create sequences
    words = poem.split()
    for i in range(1, len(words)):
        sequences.append((words[i - 1], words[i]))
```

This code feeds a dataset of poems to PyTorch and pandas. It reads in the poems, preprocesses the text to lowercase and strip punctuation, tokenizes the text into words. A vocabulary is established, and word-to-index mappings are created. It also creates a one-hot encoding function to transform word indices into vectors. Lastly, the code generates word pairs (sequences) from the poems that can be utilized for word prediction or model training. Processing is either on GPU or CPU, depending on what is available.

Preparing dataset and dataloader

```
class PoemDataset(Dataset):
    def __init__(self, sequences, word_to_idx, vocab_size):                           #
preparing dataset
```

```
        self.sequences = sequences
        self.word_to_idx = word_to_idx
        self.vocab_size = vocab_size

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

    def __getitem__(self, idx):
        x, y = self.sequences[idx]
        x_idx = self.word_to_idx[x]
        y_idx = self.word_to_idx[y]
        x_one_hot = one_hot_encode(x_idx, self.vocab_size)
        return x_one_hot, torch.tensor(y_idx, dtype=torch.long)


# Initialize dataset and dataloader
dataset = PoemDataset(sequences, word_to_idx, vocab_size)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
```

This code implements a custom dataset class PoemDataset for PyTorch usage, which is derived from Dataset. In the __init__ function, it initializes the dataset by taking in sequences (word pairs), the word_to_idx mapping, and the vocabulary size. The __len__ function returns the number of sequences, and the __getitem__ function gets a sequence at an index, converts the words to their corresponding indices, and one-hot encodes the input word (x). The target word (y) is returned as a tensor of the same index. Then, the dataset is loaded using the PoemDataset class, and a DataLoader is instantiated to manage batching and shuffling of the dataset during training. The batch_size is 128, and the data is shuffled to allow random sampling at each epoch.

Model declaration and initialization

```
class PoemGeneratorLSTM(nn.Module):
# LSTM model
    def __init__(self, vocab_size, hidden_dim, num_layers=2):
        super(PoemGeneratorLSTM, self).__init__()
        self.lstm = nn.LSTM(vocab_size, hidden_dim, num_layers=num_layers,
batch_first=True)
        self.linear = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x, hidden):
        output, hidden = self.lstm(x.unsqueeze(1), hidden)
        output = self.linear(output[:, -1, :])
        return output, hidden

# Model initialization
hidden_dim = 512
num_layers = 2
model = PoemGeneratorLSTM(vocab_size, hidden_dim, num_layers).to(device)
```

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

This code creates an LSTM-based model, PoemGeneratorLSTM, for generating poems. The model consists of an LSTM layer and a fully connected linear layer. In the **init** function, it initializes the LSTM with the vocabulary size as the input dimension, a given hidden dimension, and the number of layers (default is 2). A linear layer comes next to output the predictions for the vocabulary size.

In the forward approach, the input x (which is the one-hot representation of the word) goes through the LSTM, and its output is sent to the linear layer to generate the next word. The model also outputs the new hidden state.

The model is then initialized with a hidden dimension of 512 and 2 layers. An Adam optimizer is employed for training, with the learning rate being 0.001, and the loss function being cross-entropy loss, which is appropriate for multi-class classification problems such as predicting the next word in a sequence.

Training the model

```
epochs = 210
for epoch in range(epochs):
    total_loss = 0                                                    # training
the model
    for inputs, targets in dataloader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        hidden = None
        outputs, hidden = model(inputs, hidden)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss / len(dataloader)}")
```

This code optimizes the PoemGeneratorLSTM model for 210 epochs. During each epoch, the model propagates input and target word sequences in batches, computing loss on predicted versus actual words. The gradients of the optimizer are reset, and the parameters of the model are updated by backpropagation and the step function of the optimizer. Each batch's loss is added up, and once all batches for an epoch are processed, the loss for each batch is printed to track the learning of the model. This exercise is repeated for 210 epochs in order to improve the model's performance step by step.

Epoch 1, Loss: 7.4892905561417495
Epoch 2, Loss: 7.1605012940619275

Epoch 3, Loss: 6.4337211396410057
Epoch 4, Loss: 6.4148065710314817
‘
‘
‘
Epoch 207, Loss: 3.03347185979853
Epoch 208, Loss: 3.0317813893056287
Epoch 209, Loss: 3.030897974350292
Epoch 210, Loss: 3.032296448791583

Testing and generating the poem

```
# Generate poem function
def generate_poem_lstm(model, start_word, word_to_idx, idx_to_word, device, length=50):
    model.eval()
    generated_poem = [start_word]
    input_idx = word_to_idx[start_word]
    input_one_hot = one_hot_encode(input_idx, vocab_size).unsqueeze(0).to(device)
    hidden = None

    with torch.no_grad():
        for _ in range(length - 1):
            output, hidden = model(input_one_hot, hidden)
            probabilities = torch.softmax(output, dim=1).cpu().numpy()[0]
            predicted_word_idx = random.choices(range(len(probabilities)),
weights=probabilities, k=1)[0]
            predicted_word = idx_to_word[predicted_word_idx]
            generated_poem.append(predicted_word)
            input_one_hot = one_hot_encode(predicted_word_idx,
vocab_size).unsqueeze(0).to(device)

    formatted_poem = ""
    line = []
    word_count = 0
    for word in generated_poem:
        line.append(word)
        word_count += 1
        if word_count >= 7:
            formatted_poem += " ".join(line) + "\n"
            line = []
            word_count = 0
    if line:
        formatted_poem += " ".join(line)
    return formatted_poem

# Generate and print a poem
start_word = "sun"
generated_poem = generate_poem_lstm(model, start_word, word_to_idx, idx_to_word, device)
print("\nGenerated Poem:\n", generated_poem)
```

This code creates a function generate_poem_lstm which uses the trained LSTM model to generate a poem. The function accepts the model, a starting word, mappings from word-to-index and index-to-word, the device to be used (CPU or GPU), and the poem length to be generated. The poem is initiated with a provided word (start_word), and the function continuously predicts the next word through the model's output probabilities and chooses the next word accordingly. This step gets repeated for the given poem length.

The function also structures the generated poem by separating words into lines of seven words each, followed by a newline after each line. After the poem is generated, it is printed in a readable manner. Here, the poem begins with the word "sun" and is generated through the LSTM model. The random.choices function is utilized to sample the next word based on the output probabilities so that there is variability in the generated poem.

Output

Generated Poem:
sun there and with the calm once coalmen us notgreat
god for a while what piping heart i will believe you see
the golden same little thou frigate gale had long stretches
as to a young journey while he mile i long and reachd
breathe was the midst choir

Implementation2 (a): Trainable word embedding approach using RNN

Importing necessary libraries and framework and pre processing and tokenization

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from collections import Counter
import re
import random

df = pd.read_csv("/content/poems-100.csv")  # Loading dataset
poems = df['text'].tolist()

def clean_text(text):                       # Text preprocessing
```

```
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    return text

poems = [clean_text(poem) for poem in poems]


words = ' '.join(poems).split()                          # Tokenization
word_counts = Counter(words)
vocab = sorted(word_counts.keys())
word2idx = {word: i for i, word in enumerate(vocab, 1)}
idx2word = {i: word for word, i in word2idx.items()}
vocab_size = len(vocab) + 1


sequences = []                                          # Convert poems to sequences
seq_length = 10

for poem in poems:
    tokens = poem.split()
    for i in range(len(tokens) - seq_length):
        sequences.append([word2idx[tok] for tok in tokens[i:i + seq_length + 1]])

sequences = np.array(sequences)
X, y = sequences[:, :-1], sequences[:, -1]
```

This script initializes a data set of poems for training a language model. It begins with loading the data set from a CSV file holding poems. The clean_text function cleanses the text by turning it to lower case and removing any non-letter characters using regular expressions. This function is used to process every poem in the data set.

The code tokenizes the poems by dividing them into words and counting the frequency of each word using Counter. A vocabulary (vocab) is made by combining the unique words in a sorted way, and mappings are generated from indices to words (idx2word) and from words to indices (word2idx). The vocabulary size is found by adding 1 (for padding, which may be utilized later).

Then, the poems are transformed into fixed-length sequences (seq_length = 10). For every poem, a sliding window method is employed to create sequences of 10 words as input (X) and the subsequent word as the target (y). These sequences are saved in a numpy array and divided into the features (X) and target labels (y). The resulting information can be utilized for model training, e.g., word prediction or language generation model.

Model declaration and dataloader and hyperparamater declaration

```
class PoetryDataset(Dataset):                                  # dataset and loader
```

```
    def __init__(self, X, y):
        self.X = torch.tensor(X, dtype=torch.long)
        self.y = torch.tensor(y, dtype=torch.long)

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

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

dataset = PoetryDataset(X, y)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)


class PoetryRNN(nn.Module):                                                    # RNN
Model
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers):
        super(PoetryRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True,
nonlinearity='tanh')
        self.fc = nn.Linear(hidden_dim, vocab_size)

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

# Hyperparameters
embed_dim = 128
hidden_dim = 256
num_layers = 10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

This code creates a custom dataset class PoetryDataset for handling and loading poetry data, to be used with PyTorch's DataLoader. The dataset class accepts input features (X) and target labels (y), transposes them into PyTorch tensors, and has methods for accessing data. It specifies the **len** method to get the number of samples and the **getitem** method to get a particular sample by index.

The PoetryRNN class specifies an RNN model for poetry generation. It has three layers:

Embedding Layer: Maps word indices to dense vectors of a given size (embed_dim).

RNN Layer: A recurrent neural network that processes the embedded input. The number of hidden units is set by hidden_dim, and the number of layers by num_layers. The nonlinearity employed in the RNN is 'tanh'.

Fully Connected Layer: A linear layer giving predictions for every word in the vocabulary. The forward method of the model goes through the sequence of input by passing it through the embedding and RNN layers and returns the predicted word at the final time step, as well as the new hidden state.

The following hyperparameters are used:

Embedding dimension: 128 Hidden dimension: 256 Number of RNN layers: 10 Device: The model will execute on GPU if available, else on the CPU. This model can now be trained with the `PoetryDataset` and `DataLoader` defined for batch processing efficiently.

Model initialization and training

```
model = PoetryRNN(vocab_size, embed_dim, hidden_dim, num_layers).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Training
def train_model(epochs=180):
    model.train()
    for epoch in range(epochs):
        for X_batch, y_batch in dataloader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)


            hidden = torch.zeros(num_layers, X_batch.size(0), hidden_dim).to(device)
# moving data to gpu

            optimizer.zero_grad()
            output, hidden = model(X_batch, hidden)
            loss = criterion(output, y_batch)
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")

train_model()
```

This code trains the PoetryRNN model for generating poetry. The model is initialized with given hyperparameters, such as vocabulary size, embedding dimension, hidden dimension, and number of layers, and is transferred to the correct device (GPU or CPU). CrossEntropyLoss function is utilized for loss, and Adam optimizer is used to update the parameters of the model. The train_model function trains for 180 epochs (default), with in each epoch the model iterating over batches from the dataloader. It calculates the loss, does backpropagation, and updates the weights of the model as a result. The training loop outputs the loss per epoch, demonstrating how the model is improving its prediction of the next word in a sequence over epochs.

Epoch 1/180, Loss: 6.3788
Epoch 2/180, Loss: 7.1956
Epoch 3/180, Loss: 7.1874
‘
‘
‘
Epoch 178/180, Loss: 7.0970
Epoch 179/180, Loss: 7.0223
Epoch 180/180, Loss: 7.0358

Poem generation

```
import torch.nn.functional as F
def generate_poem(seed_text, max_words=50, line_length=7):
    model.eval()
#evaluation mode
    words = seed_text.lower().split()
    generated_poem = words[:]
# Store generated words
    hidden = torch.zeros(num_layers, 1, hidden_dim).to(device)
# Initialize hidden state

    for _ in range(max_words):
        # Convert words to indices
        input_seq = [word2idx.get(word, 0) for word in words[-seq_length:]]
        input_tensor = torch.tensor(input_seq, dtype=torch.long).unsqueeze(0).to(device)
# Generate next word
        with torch.no_grad():
            output, hidden = model(input_tensor, hidden)
            predicted_idx = torch.argmax(output, dim=1).item()
        predicted_word = idx2word.get(predicted_idx, "<UNK>")
        generated_poem.append(predicted_word)
        if predicted_word == "<eos>":
            break
    poem_lines = [" ".join(generated_poem[i:i+line_length]) for i in range(0,
len(generated_poem), line_length)]
    return "\n".join(poem_lines)

# Example Usage
print(generate_poem("Sun"))
```

The code creates a generate_poem function that creates a poem from a seed_text of a given length. The function puts the model into evaluation mode and initializes the hidden state. It then iteratively predicts the next word by converting the previous seq_length words into indices, passing them through the model, and choosing the word with the highest predicted probability. This goes on until the maximum word count is hit or an end-of-sequence token () is estimated. The generated words are placed in lines of a given line

length and given as a finished poem. The example is illustrated where the function creates a poem beginning with "Sun."

Output

sun the and the and the and the and the and the
and the and the and the and the and the and the
and the and the and the and the and the and the
and the and the and the and the and the and the
and the and

Implementation2 (b) : Trainable word embedding approach using LSTM

Importing libraries and framework and preprocessing

```
import torch
import torch.nn as nn
import torch.optim as optim
import pandas as pd
import numpy as np
from torch.utils.data import Dataset, DataLoader
import re

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# Check for GPU availability
print(f"Using device: {device}")

df = pd.read_csv("/content/poems-100.csv")
# Load the CSV file
poems = df["text"].tolist()

def preprocess_text(text):
# Preprocess the poems
    text = re.sub(r'[^\w\s]', '', text).lower()
    return text
processed_poems = [preprocess_text(poem) for poem in poems]
```

This code sets up the environment and data for training a poetry generation model. It checks first whether a GPU can be used for training and sets the device. The poems are read from a CSV file (poems-100.csv) into a pandas DataFrame, and the text column is read into a list of poems. The preprocess_text function is declared to preprocess every poem by stripping off punctuation and converting all characters to lowercase to ensure

consistency in the data. The poems are then preprocessed with this function, and the cleaned poems are ready for additional processing, like tokenization and model training.

Tokenization and sequencing

```
words = []
#tokenization
for poem in processed_poems:
    words.extend(poem.split())

vocab = sorted(list(set(words)))
word_to_idx = {w: i for i, w in enumerate(vocab)}
idx_to_word = {i: w for i, w in enumerate(vocab)}

vocab_size = len(vocab)


sequence_length = 10
sequences = []                                                    #
creating sequences
next_words = []

for poem in processed_poems:
    poem_words = poem.split()
    if len(poem_words) > sequence_length:
        for i in range(len(poem_words) - sequence_length):
            seq = poem_words[i:i + sequence_length]
            next_word = poem_words[i + sequence_length]
            sequences.append([word_to_idx[word] for word in seq])
            next_words.append(word_to_idx[next_word])

sequences = torch.tensor(sequences, dtype=torch.long).to(device)
next_words = torch.tensor(next_words, dtype=torch.long).to(device)
```

This code tokenizes the preprocessed poems by dividing each poem into words and building a vocabulary of distinctive words. It then constructs mappings from word to index (word_to_idx) and from index to word (idx_to_word). The vocab size (vocab_size) is computed from the number of distinctive words.

Then, word sequences are created for model training. A sliding window method is employed to form sequences of 10 words (sequence_length), followed by the next word in each sequence. The word sequences and next words are then translated into indices with the word_to_idx dictionary. Lastly, the sequences and next words are translated into PyTorch tensors and transferred to the target device (GPU or CPU) for model training.

Dataset and dataloader

```
                                                                    #
Dataset and DataLoader
class PoemDataset(Dataset):
    def __init__(self, sequences, next_words):
        self.sequences = sequences
        self.next_words = next_words

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

    def __getitem__(self, idx):
        return self.sequences[idx], self.next_words[idx]

dataset = PoemDataset(sequences, next_words)
dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
```

This code creates a custom dataset class PoemDataset for handling the sequences of words and their next words. The class is a subclass of PyTorch's Dataset and needs the sequences of words and next words as parameters. The **len** method returns the length of the sequences, and the **getitem** method gets a sequence and its next word by index.

The data is then utilized to form a DataLoader, which batches the training data. The batch_size is 128, and the data is shuffled prior to each epoch so that the model will not learn any unwanted patterns from the sequence order. This configuration enables efficient batching and loading of the data during model training.

Model declaration , initialization and hyperparameter tuning

```
class PoemGenerator(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_layers):
        super(PoemGenerator, self).__init__()
# LSTM Model
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x):
        embedded = self.embedding(x)
        lstm_out, _ = self.lstm(embedded)
        output = self.fc(lstm_out[:, -1, :])
        return output

# Hyperparameters
embedding_dim = 256
hidden_dim = 512
num_layers = 2
learning_rate = 0.001
epochs = 50

# Initialize the model
```

```
model = PoemGenerator(vocab_size, embedding_dim, hidden_dim, num_layers).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

This code is implementing a PoemGenerator class, an LSTM model for generating poems. The model has three primary components:

Embedding Layer: Maps the word indices to high-dimensional dense vectors of a given size (embedding_dim). LSTM Layer: Handles the embedded input and generates a hidden state of dimension hidden_dim. It has num_layers layers, which enables it to learn sophisticated temporal dependences in the input sequences. Fully Connected Layer: Linear layer that provides a prediction for every word in the vocabulary. The forward method passes the input through the embedding layer, LSTM, and fully connected layer to return the last time step predicted word.

Model hyperparameters are defined:

Embedding dimension: 256

Hidden dimension: 512

Number of LSTM layers: 2

Learning rate: 0.001

Epochs: 50

The model is then initialized and transferred to the respective device (GPU or CPU). The loss function employed is CrossEntropyLoss, which is appropriate for multi-class classification problems such as predicting the next word in the sequence, and the Adam optimizer is employed for training.

Training the model

```
# Training loop
for epoch in range(epochs):
    for sequences_batch, next_words_batch in dataloader:
        optimizer.zero_grad()
        outputs = model(sequences_batch)
        loss = criterion(outputs, next_words_batch)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item()}")
```

The training loop continues for a given number of epochs (50 here). In each epoch, the model takes batches of sequences and the respective next words from the dataloader. For

every batch, the model does a forward pass to make predictions (outputs) from the sequence inputs. The loss is computed by CrossEntropyLoss between the model's predictions and the actual next words (next_words_batch). The gradients are subsequently computed by using loss.backward() and the optimizer updates the parameters of the model by using optimizer.step(). Each epoch, the loss is also printed to keep track of how the model's training is proceeding. This will enable the model to learn its poetry generation gradually by reducing the loss function through time.

Epoch 1/50, Loss: 6.881255149841309
Epoch 2/50, Loss: 6.124566078186035
Epoch 3/50, Loss: 5.903942584991455
‘
‘
‘
Epoch 48/50, Loss: 0.0005366378463804722
Epoch 49/50, Loss: 0.0004747148195747286
Epoch 50/50, Loss: 0.0004367862129583955

Poem generation and evaluation

```
import random
def generate_poem(model, start_word, word_to_idx, idx_to_word, device, length=50):
    model.eval()
    generated_poem = [start_word]
    input_seq = torch.tensor([[word_to_idx[start_word]]], dtype=torch.long).to(device)

    with torch.no_grad():
        for _ in range(length - 1):
            output = model(input_seq)
            probabilities = torch.softmax(output, dim=1).cpu().numpy()[0] #get
probability distribution
            predicted_word_idx = random.choices(range(len(probabilities)),
weights=probabilities, k=1)[0] # sample from the distribution.
            predicted_word = idx_to_word[predicted_word_idx]
            generated_poem.append(predicted_word)
            input_seq = torch.tensor([[predicted_word_idx]],
dtype=torch.long).to(device)


    formatted_poem = ""                                    #  Format the poem into
multiple lines
    line = []
    word_count = 0
    for word in generated_poem:
        line.append(word)
        word_count += 1
        if word_count >= 7:
            formatted_poem += " ".join(line) + "\n"
            line = []
```

```
            word_count = 0
    if line:  # Add any remaining words
        formatted_poem += " ".join(line)
    return formatted_poem


start_word = "city"
generated_poem = generate_poem(model, start_word, word_to_idx, idx_to_word, device)
print("Generated Poem:\n", generated_poem)
```

The `generate_poem` function generates a poem starting from a given `start_word` using the trained `PoemGenerator` model. It begins by setting the model to evaluation mode, ensuring that dropout and other training-specific behaviors are disabled. The function converts the `start_word` into an index using the `word_to_idx` dictionary and prepares it as input for the model. In each iteration of the loop, the model predicts the next word, which is sampled from the predicted probability distribution using softmax. The generated word is added to the poem, and the input for the next prediction becomes the previously predicted word. After generating the specified number of words, the poem is formatted into lines of no more than seven words. The function then returns the formatted poem, providing a coherent, contextually relevant piece of poetry starting from the seed word. In the example, the poem generation starts with the word "city."

Output

Generated Poem:
city the candle tickets take anchor whirr or words
colter commanding bends a turnpike rye seat trapper in words
that nothing has retiring see shaken breed call health swimmers
dates say mortars nudge vines angry mould completes the hairy
destiny beg companion betwixt four autumn keep volumes spread lit muscular

COMPARISON:

We will focus on RNN(LSTM) model for both approaches as vanilla RNN with trainable word embedding approach is unfortunately flawed in this case.

S.no	One hot encodding approach	Trainable word embedding approach
1)	Training loss after 210 epoch = 0.30332964 Higher loss even after 210 epoch	Training loss after only 50 epochs = 0.0004367 Lower loss only after 50 epochs
2)	Time taken to complete all epoch during training with batch size of 128 ~ 26 minutes. High training time	Time taken to complete all epoch during training with batch size of 128 ~ 9 minutes. Lower training time
3)	Total memory usage= 3.2GB/12.5GB for T4 gpu (available in colab) Higher memory usage	Total memory usage = 2.1GB/12.5GB for T4 GPU (available in T4) Lower memory usage
4)	Output city the candle tickets take anchor whirr or words colter commanding bends a turnpike rye seat trapper in words that nothing has retiring see shaken breed call health swimmers dates say mortars nudge vines angry mould completes the hairy destiny beg companion betwixt four autumn keep volumes spread lit muscular	Output Generated Poem: sun there and with the calm once coalmen us notgreat god for a while what piping heart i will believe you see the golden same little thou frigate gale had long stretches as to a young journey while he mile i long and reachd breathe was the midst choir of

<u>CONCLUSION</u>:

LSTM RNN model is recommended for poem generation purpose with “trainable words embedding approach”. Not only will it converge much more quickly , but also will take less time and memory footprint.

Hence , Poem generation is performed using RNNs with “one hot encoding approach” and “trainable word embedding approach”.