Character-Based Text Generation

Davide Giuseppe Griffon

Abstract

This document serves as the report for the fourth task in the "Natural Language Processing" course completed by student Davide Giuseppe Griffon at Vilnius University as part of the Master's program in Data Science.

Contents

1 Introduction

In this project, I developed a character-based text generation system using Long Short-Term Memory (LSTM) networks. The model was trained on a collection of works by G.K. Chesterton from the Gutenberg corpus, with the goal of generating new text that mimics the author's writing style.

A text corpus is a large, structured collection of texts that serves as a foundational resource in natural language processing (NLP). Text corpora provide the raw material necessary for analyzing language patterns, training models, and evaluating their performance. In NLP applications, corpora are essential because they offer authentic examples of language usage, allowing models to learn naturally occurring patterns rather than relying on manually coded rules.

The remainder of this document is organized as follows. First, I describe the data loading and preprocessing steps, including the creation of character encodings. Then, I detail the architecture of the LSTM model and the training process. Finally, I present the results of text generation experiments and discuss potential improvements to the system.

2 Text generation

2.1 Loading the Data

For this project, I utilized Chesterton's works available in the NLTK Gutenberg corpus, which comprises three major books: "The Ball and the Cross", "The Wisdom of Father Brown", and "The Man Who Was Thursday". To facilitate data loading and preprocessing, I implemented a `GutenbergLoader` class with various utility functions.

The corpus analysis revealed the following statistics for each work:

Work	**Characters**	**Words**	**Lines**
The Ball and the Cross	457,450	81,598	9,548
Father Brown	406,629	71,626	7,654
The Man Who Was Thursday	320,525	57,955	6,793
Total	1,184,604	211,179	23,995

Table 1: Statistics of Chesterton's works in the corpus

The combined corpus consists of 1,184,604 characters in total, with a vocabulary of 91 unique characters. The complete character set includes:

```
! " $ % ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] _ ‘
a b c d e f g h i j k l m n o p q r s t u v w x y z ~ è é î
```

2.2 One-hot Encoding and Decoding

For this purpose, I wrote the `CharacterEncoder` class which is responsible for encoding and decoding characters using one-hot encoding. The method `fit` creates bidirectional mappings through two simple dictionary comprehensions:

```python
self.char_to_index = {
    char: idx for idx, char in enumerate(unique_chars)
}
self.index_to_char = {
    idx: char for idx, char in enumerate(unique_chars)
}
```

The first line creates a mapping from characters to indices, while the second creates the reverse mapping. Using Python's `enumerate` function, each unique character is assigned a consecutive integer index, making the encoding and decoding process straightforward.

Furthermore, methods `save_mappings` and `load_mappings` are responsible for saving and loading these mappings from a JSON file, making the loading process more efficient during subsequent steps.

2.3 Building the Training Dataset

To train the character-based language model, I needed to create overlapping sequences from the input text. For this purpose, I implemented the `TextDataset` class, which inherits from PyTorch's `Dataset` class. Each sequence in the training set consists of 40 characters (`self.seq_length = 40`), and the target is the character that follows this sequence.

The core functionality resides in the `__getitem__` method:

```python
def __getitem__(self, idx: int) -> tuple[torch.Tensor, torch.Tensor]:
    # Get the sequence and target
    sequence = self.text_indices[idx : idx + self.seq_length]
    target = self.text_indices[idx + self.seq_length]

    # Create one-hot encoding for input sequence
    X = torch.zeros(
        self.seq_length, self.char_encoder.vocab_size, dtype=torch.
           float32
    )
    for t, char_idx in enumerate(sequence):
        X[t, char_idx] = 1

    # Return target as a single integer (not one-hot encoded)
    return X, torch.tensor(target, dtype=torch.long)
```

This method creates overlapping sequences by sliding a window of 40 characters over the text. For each sequence, it returns:

- A one-hot encoded tensor of the input sequence with shape `(40, vocab_size)`
- The index of the target character (the 41st character) as a single integer

2.4 LSTM Architecture

For this project, I implemented a simple LSTM (Long Short-Term Memory) network using PyTorch. The model architecture consists of two main components:

```python
class LSTM(nn.Module):
    def __init__(self, input_size: int, hidden_size: int = 128):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
```

```
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers=1,
            batch_first=True)
        self.fc = nn.Linear(hidden_size, input_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x, _ = self.lstm(x)
        # Return logits (softmax will be applied in the generation step
            )
        return self.fc(x[:, -1, :])
```

The architecture consists of:

- A single LSTM layer with a hidden size of 128 units
- A fully connected layer that maps the LSTM's output back to the vocabulary size

The model takes a sequence of one-hot encoded characters as input and outputs raw logits over the possible next characters. The softmax function is not applied in the forward pass but rather during the text generation phase to convert these logits into probabilities.

2.5 Performance Evaluation

The generation of text is performed using a proxy class `TextGenerator` that loads the trained model and generates text using the `generate_text` method. The diversity parameter controls the randomness in the text generation process by scaling the logits before applying the softmax function.

The code for text generation is as follows:

```
seed_text = (
    "There was an instant of rigid silence, and then Syme in his turn "
    "fell furiously on the other, filled with a flaming curiosity."
)[:40]
# Note that the truncated text is:
# "There was an instant of rigid silence, a"
for diversity in [0.2, 0.5, 1.0]:
    print(f"\nDiversity: {diversity}")
    generated = generator.generate_text(
        seed_text=seed_text, length=100, diversity=diversity
    )
    print(generated)
```

Here are examples of generated text with different diversity levels:

- **Diversity: 0.2**
  "There was an instant of rigid silence, and the beard and the black strange of the streets of the street and face and strong and streets that"
- **Diversity: 0.5**
  "There was an instant of rigid silence, and in the two some light and sunset the chair and sat desportably entirely like a strong face and be"
- **Diversity: 1.0**
  "There was an instant of rigid silence, and the writity-slack't read and gatening silks, by might began tomprecy that they have then heally e"

The results demonstrate several limitations of the character-level model:

- Despite occasionally generating existing words, the overall text lacks coherence and semantic meaning
- The model shows no understanding of broader context or grammar
- Higher diversity levels (`diversity = 1.0`) lead to the generation of non-existent words and more random text sequences
- Lower diversity levels (`diversity = 0.2`) produce more repetitive patterns but still lack meaningful structure

These limitations are inherent to the character-level approach, as the model operates without any understanding of word-level semantics or linguistic structure. While it can learn character patterns and combinations common in English text, it cannot capture higher-level language features that would be necessary for generating coherent narratives.

2.6 Challenges and Possible Improvements

One of the main challenges I encountered during this project was the significant computational time required for the model training phase. This slowness stems from the inherent complexity of processing large volumes of character-level data combined with the computational demands of the model architecture. While I utilized the MPS (Metal Performance Shaders) device available on my Apple machine, which provided better performance than CPU processing, the lack of GPU (CUDA from NVIDIA) access remained a limiting factor in achieving optimal training speeds. I attempted to address these performance issues by experimenting with a reduced dataset, using only one book from Chesterton, but this approach resulted in notably worse generation quality. Increasing the batch size provided some improvement in training speed, though the overall process remained time-consuming.

The limitations in the generated text quality can be attributed primarily to the character-level approach of the model. By processing text one character at a time and only considering the previous 40 characters for context, the model struggles to grasp higher-level linguistic concepts such as word meanings, grammatical structures, and broader contextual relationships. This limitation manifests in the generated text as a lack of coherence and meaningful narrative flow, even though the model can sometimes produce valid English words.

Looking toward potential improvements, several things could enhance both the model's performance and the quality of generated text. The most straightforward enhancement would be implementing the training process on a GPU, which would significantly reduce training time and allow to train more epochs in fewer time. From an architectural perspective, transitioning to a word-level model could provide better semantic understanding and more coherent text generation. Additionally, implementing a transformer architecture, which has demonstrated impressive results in various text generation tasks, could significantly improve the quality of the generated content. The model could also benefit from an expanded training dataset incorporating more works from Chesterton and/or other authors, providing a richer foundation for learning language patterns and improving the overall generation capabilities.

3 Appendix - Code

All the code is available in the GitHub repository `https://github.com/Griffosx/nlp` under the src/task_4 folder. For completeness, I include here the code for the main files used in this project.

File loader.py

```python
import nltk
from nltk.corpus import gutenberg


class GutenbergLoader:
    def __init__(self):
        # Ensure the Gutenberg dataset is downloaded
        nltk.download("gutenberg")

    def list_available_works(self) -> list[str]:
        """List all available works in the Gutenberg corpus."""
        return gutenberg.fileids()

    def list_author_works(self, author: str) -> list[str]:
        """List all works by a specific author."""
        return [
            work
            for work in self.list_available_works()
            if author.lower() in work.lower()
        ]

    def load_chesterton_works(self) -> dict[str, str]:
        """Load Chesterton's major works from the Gutenberg corpus."""
        chesterton_works = {
            "ball": "chesterton-ball.txt",
            "brown": "chesterton-brown.txt",
            "thursday": "chesterton-thursday.txt",
        }

        loaded_works = {}
        for title, filename in chesterton_works.items():
            try:
                text = gutenberg.raw(filename)
                loaded_works[title] = text
                print(f"Successfully loaded: {filename}")
            except Exception as e:
                print(f"Error loading {filename}: {str(e)}")

        return loaded_works

    def get_combined_text(self, works: dict[str, str]) -> str:
        """Combine all loaded works into a single text."""
        return "\n\n".join(works.values())

    def get_chesterton_combined_text(self) -> str:
        """Load and combine Chesterton's major works."""
        works = self.load_chesterton_works()
        return self.get_combined_text(works)

    def print_corpus_stats(self, works: dict[str, str]) -> None:
```

```
        """Print basic statistics about the loaded works."""
        print("\nCorpus Statistics:")
        total_chars = 0
        total_words = 0
        total_lines = 0

        for title, text in works.items():
            num_chars = len(text)
            num_words = len(text.split())
            num_lines = len(text.splitlines())

            total_chars += num_chars
            total_words += num_words
            total_lines += num_lines

            print(f"\n{title.title()}:")
            print(f"Characters: {num_chars:,}")
            print(f"Words: {num_words:,}")
            print(f"Lines: {num_lines:,}")

        print("\nTotal Statistics:")
        print(f"Total Characters: {total_chars:,}")
        print(f"Total Words: {total_words:,}")
        print(f"Total Lines: {total_lines:,}")

    def show_unique_characters(self, text: str) -> None:
        """Display all unique characters in the text."""
        unique_chars = sorted(list(set(text)))
        print("\nUnique Characters in Corpus:")
        print(f"Number of unique characters: {len(unique_chars)}")
        print("Characters: ", " ".join(unique_chars))


def print_chesterton_info():
    # Initialize the loader
    loader = GutenbergLoader()

    # List all Chesterton works available
    print("Available Chesterton works in Gutenberg:")
    chesterton_works = loader.list_author_works("chesterton")
    for work in chesterton_works:
        print(f"- {work}")

    # Load Chesterton's works
    print("\nLoading Chesterton's works...")
    works = loader.load_chesterton_works()

    # Print statistics for each work
    loader.print_corpus_stats(works)

    # Get combined text
    combined_text = loader.get_combined_text(works)

    # Show unique characters
    loader.show_unique_characters(combined_text)
```

File encoder.py

```
import json
```

```python
from loader import GutenbergLoader


class CharacterEncoder:
    def __init__(self):
        self.char_to_index: dict[str, int] = {}
        self.index_to_char: dict[int, str] = {}
        self.vocab_size: int = 0

    def fit(self, text: str) -> None:
        """Create character to index mappings from text."""
        # Get unique characters and sort them for consistency
        unique_chars = sorted(list(set(text)))
        self.vocab_size = len(unique_chars)

        # Create bidirectional mappings
        self.char_to_index = {char: idx for idx, char in enumerate(
            unique_chars)}
        self.index_to_char = {idx: char for idx, char in enumerate(
            unique_chars)}

    def save_mappings(self, filename: str) -> None:
        """Save character mappings to a JSON file."""
        mappings = {
            "char_to_index": self.char_to_index,
            "index_to_char": {
                str(k): v for k, v in self.index_to_char.items()
            },  # Convert int keys to str for JSON
            "vocab_size": self.vocab_size,
        }
        with open(filename, "w", encoding="utf-8") as f:
            json.dump(mappings, f, ensure_ascii=False, indent=2)

    def load_mappings(self, filename: str) -> None:
        """Load character mappings from a JSON file."""
        with open(filename, "r", encoding="utf-8") as f:
            mappings = json.load(f)
        self.char_to_index = mappings["char_to_index"]
        self.index_to_char = {
            int(k): v for k, v in mappings["index_to_char"].items()
        }  # Convert str keys back to int
        self.vocab_size = mappings["vocab_size"]

    def print_mappings(self) -> None:
        """Print character mappings in a readable format."""
        print("\nCharacter to Index Mappings:")
        print("-" * 30)
        for char, idx in sorted(self.char_to_index.items(), key=lambda
            x: x[1]):
            if char.isspace():
                char_display = f"[space-{ord(char)}]"
            elif char == "\n":
                char_display = "[newline]"
            elif char == "\t":
                char_display = "[tab]"
            else:
                char_display = char
            print(f"'{char_display}' -> {idx}")
```

```
        print(f"\nTotal vocabulary size: {self.vocab_size}")
```

File generator.py

```
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from task_4.loader import GutenbergLoader
from task_4.encoder import CharacterEncoder


class TextDataset(Dataset):
    def __init__(self, text: str, char_encoder: CharacterEncoder, seq_length: int):
        self.text = text
        self.char_encoder = char_encoder
        self.seq_length = seq_length
        self.text_indices = [char_encoder.char_to_index[char] for char in text]
        self.num_sequences = len(self.text_indices) - self.seq_length - 1

    def __len__(self) -> int:
        return self.num_sequences

    def __getitem__(self, idx: int) -> tuple[torch.Tensor, torch.Tensor]:
        # Get the sequence and target
        sequence = self.text_indices[idx : idx + self.seq_length]
        target = self.text_indices[idx + self.seq_length]

        # Create one-hot encoding for input sequence
        X = torch.zeros(
            self.seq_length, self.char_encoder.vocab_size, dtype=torch.float32
        )
        for t, char_idx in enumerate(sequence):
            X[t, char_idx] = 1

        # Return target as a single integer (not one-hot encoded)
        return X, torch.tensor(target, dtype=torch.long)


class LSTM(nn.Module):
    def __init__(self, input_size: int, hidden_size: int = 128):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers=1, batch_first=True)
        self.fc = nn.Linear(hidden_size, input_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x, _ = self.lstm(x)
        # Return logits (softmax will be applied in the generation step)
        return self.fc(x[:, -1, :])


```

```python
class TextGenerator:
    def __init__(
        self,
        seq_length: int = 40,
        device: str = "mps",
    ):
        self.seq_length = seq_length
        self.device = device
        self.model = None

    def build_model(self, vocab_size: int) -> None:
        """Construct the LSTM model."""
        self.model = LSTM(vocab_size).to(self.device)

    def train(
        self,
        text: str,
        char_encoder: CharacterEncoder,
        model_save_path: str = "model.pth",
        encoder_save_path: str = "char_encoder.json",
        epochs: int = 10,
        batch_size: int = 32,
        learning_rate: float = 0.001,
    ) -> list[float]:
        """Train the model and save it to disk."""
        if self.model is None:
            self.build_model(char_encoder.vocab_size)

        dataset = TextDataset(text, char_encoder, self.seq_length)
        dataloader = DataLoader(dataset, batch_size=batch_size, shuffle
            =True)

        # CrossEntropyLoss expects raw logits and target indices
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.RMSprop(self.model.parameters(), lr=
            learning_rate)

        losses = []
        self.model.train()

        for epoch in range(epochs):
            epoch_loss = 0
            batch_count = 0

            for batch_X, batch_y in dataloader:
                batch_X = batch_X.to(self.device)
                batch_y = batch_y.to(self.device)

                optimizer.zero_grad()
                # Get logits from model
                logits = self.model(batch_X)
                # CrossEntropyLoss expects logits and target class
                    indices
                loss = criterion(logits, batch_y)

                loss.backward()
                optimizer.step()

```

```
                epoch_loss += loss.item()
                batch_count += 1

                del batch_X, batch_y, logits, loss

                if batch_count % 100 == 0:
                    print(f"Epoch {epoch+1}, Batch {batch_count}/{len(dataloader)}")

            avg_loss = epoch_loss / batch_count
            losses.append(avg_loss)
            print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")

        # Save model and encoder
        torch.save(self.model.state_dict(), model_save_path)
        char_encoder.save_mappings(encoder_save_path)
        print(f"Model saved to {model_save_path}")
        print(f"Character encoder saved to {encoder_save_path}")

        return losses

    def generate_text(
        self,
        seed_text: str,
        model_load_path: str = "model.pth",
        encoder_load_path: str = "char_encoder.json",
        length: int = 100,
        diversity: float = 0.5,
    ) -> str:
        """Load model and generate text."""
        char_encoder = CharacterEncoder()
        char_encoder.load_mappings(encoder_load_path)

        if self.model is None:
            self.build_model(char_encoder.vocab_size)
            self.model.load_state_dict(
                torch.load(model_load_path, map_location=self.device, weights_only=True)
            )

        if len(seed_text) != self.seq_length:
            raise ValueError(f"Seed text must be {self.seq_length} characters long")

        self.model.eval()
        current_sequence = seed_text
        generated_text = seed_text

        with torch.no_grad():
            for _ in range(length):
                # Prepare input tensor
                x_pred = torch.zeros(
                    (1, self.seq_length, char_encoder.vocab_size), dtype=torch.float32
                )
                for t, char in enumerate(current_sequence):
                    if char in char_encoder.char_to_index:
                        x_pred[0, t, char_encoder.char_to_index[char]]
```

```
                           = 1

                # Generate prediction
                x_pred = x_pred.to(self.device)
                # Get logits from model
                logits = self.model(x_pred)

                # Apply softmax and temperature scaling for generation
                preds = torch.softmax(logits / diversity, dim=-1).cpu().numpy()[0]

                # Sample next character
                next_index = np.random.choice(len(preds), p=preds)
                next_char = char_encoder.index_to_char[next_index]

                # Update sequences
                generated_text += next_char
                current_sequence = current_sequence[1:] + next_char

                del x_pred, logits

        return generated_text


def train():
    loader = GutenbergLoader()
    encoder = CharacterEncoder()
    generator = TextGenerator(seq_length=40)

    combined_text = loader.get_chesterton_combined_text()
    encoder.fit(combined_text)

    epochs = 10
    batct_size = 512
    losses = generator.train(
        text=combined_text, char_encoder=encoder, epochs=epochs, batch_size=batct_size
    )
    print("Training completed!")


def generate():
    generator = TextGenerator(seq_length=40)

    seed_text = (
        "There was an instant of rigid silence, and then Syme in his turn fell "
        "furiously on the other, filled with a flaming curiosity."
    )[:40]
    print(f"Seed text: {seed_text}")

    print("\nSeed text:")
    print(seed_text)
    print("\nGenerated texts with different diversity levels:")

    for diversity in [0.2, 0.5, 1.0]:
        print(f"\nDiversity: {diversity}")
        generated = generator.generate_text(
```

```
            seed_text=seed_text, length=100, diversity=diversity
        )
        print(generated)
```