Assignment 4: Character-Based Text Generation

Aleksandr Jan Smoliakov

2024–12–17

1 Introduction

In this project, my objective is to build a machine learning model that can generate natural language text. To achieve this, I will take a character-based generation approach, where the model learns to predict the next character on a sequence of 40 preceding characters. Longer sequences will be produced by generating one character at a time and appending it to the input sequence for the next prediction.

The project will involve several steps, including data preparation, model training, and text generation. I will use Jane Austen's works as the training data and train an LSTM (Long Short-Term Memory) neural network to learn the language patterns present in her writing. Once the model is trained, I will generate text samples using different diversity parameters to control the creativity and coherence of the generated text.

Finally, I will evaluate the generated text qualitatively and discuss the limitations of the model and potential improvements for future work.

2 Methodology

2.1 Dataset

Natural Language Processing often relies on large collections of text known as corpora (plural of corpus). A **text corpus** is essentially a large set of textual data. In NLP, text corpora serve as the foundation for training and evaluating language models and other language-based algorithms. By feeding text corpora to machine learning models, we enable them to learn the statistical patterns and structures inherent in natural language. The models then leverage these learned patterns to perform tasks such as text generation, sentiment analysis, machine translation, and more.

In this project, I chose to train the model on a subset of the *Gutenberg Corpus*. This corpus contains a wide range of texts from the Project Gutenberg, an archive of over 70,000 free eBooks, including works by classical authors such as William Shakespeare, Jane Austen, and others. The texts in this corpus are widely used as benchmark datasets for NLP tasks. The Python *NLTK* library

provides easy access to a subset of the Gutenberg Corpus, making it a convenient choice.

For this project, I selected three works by Jane Austen:

- "Emma"
- "Persuasion"
- "Sense and Sensibility"

The combined text from these novels will serve as the training data for the character-based text generation model.

2.2 Data Preprocessing

Before training the model, the following preprocessing steps were applied to the text data.

First, the texts were loaded and concatenated into a single string. The total number of characters in this combined text was 2.03 million, with 82 unique characters, including letters, punctuation, whitespace, and digits.

Additionally, to make the task more manageable for the model, I converted all characters to lowercase. This step helps reduce the vocabulary size (in our case, the alphabet size) and ensures that the model does not treat uppercase and lowercase versions of the same character as distinct entities. This way, the number of unique characters was reduced from 82 to 56.

2.3 Character-to-Index Mapping

The language model does not work with characters directly; as a classification model, it works with classes that are commonly represented by their integer indices. To enable the model to handle characters, I needed to translate them into numeric form suitable for the machine learning model. This was achieved by first creating a two-way mapping between characters and integers that represent 'character indices'.

I extracted the set of unique characters that appeared in the corpus. Each unique character was then assigned a unique integer index. The 56 unique characters were mapped as follows:

```
'\n': 0,
' ': 1,
'!': 2,
...,
'‘': 29,
'a': 30,
'b': 31,
...,
'z': 55,
```

This dictionary allows transforming any character into a numerical index.

To achieve the reverse mapping (from index to character), I created a second dictionary that maps the integer indices back to their corresponding characters:

```
0: '\n',
1: ' ',
2: '!',
...,
29: '‘',
30: 'a',
31: 'b',
...,
55: 'z',
```

These dictionaries were used for one-hot encoding and decoding characters during the training and text generation processes.

2.4 Overlapping Sequence Generation

The objective of the model is to predict the next character in a sequence of characters. To train the model, I needed to prepare a dataset of input-output pairs. This was achieved by sampling overlapping sequences of characters from the text data.

I chose an input sequence length of 40 characters, meaning that each training sample consisted of 40 consecutive characters from the text. By sliding a window of size 40 characters through the combined text, I created many overlapping sequences. For every 40-character sequence (X), the next character (y) was recorded as the "label" to predict. This process was repeated by moving the window three characters forward to create the next training sample, until the end of the text was reached.

This overlapping strategy ensures that the model is exposed to the entire text data. Ideally, I would have used a step size of 1 to create even more training samples and use the maximum amount of data available in the dataset. However, this would have significantly increased the size of the dataset and training time.

As an example of how this works, consider the following snippet from the text:

```
"highbury, the large and populous village, almost amounting to a town,
to which hartfield, in spite of its separate lawn"
```

40-character windows and their corresponding next characters are generated as follows:

```
"highbury, the large and populous village" -> ","
"hbury, the large and populous village, a" -> "l"
"ry, the large and populous village, almo" -> "s"
...
```

The 41st character (the one immediately following this sequence) is what we want the model to predict. Using the above approach, I created about 675 thousand of such overlapping training samples.

2.5 Model Architecture

LSTM (Long Short-Term Memory) networks are a type of recurrent neural network (RNN) designed to work with sequential data. They are equipped with memory cells and input, forget, and output gates to handle the vanishing gradient problem and capture long-term dependencies. They are well-suited for language modeling tasks, as they can retain information over many time steps and recognize patterns in sequences.

The neural network architecture was built using the PyTorch library. The model I used for this project consists of the following components:

- **Input:** One-hot encoded characters are fed into the model. The input shape is $(sequence\_length, number\_of\_unique\_characters)$.
- **LSTM Layers:** Two LSTM layers.
  - A bi-directional LSTM layer with 64 units in each direction, followed by dropout=0.2 (to prevent overfitting).
  - Another bi-directional LSTM layer with 64 units in each direction.
- **Dense Output Layer:** A fully connected (Dense) layer with 56 units (56 being the number of unique characters) and a softmax activation. The softmax ensures that the output can be interpreted as a probability distribution over all possible next characters.

2.6 Training the Model

The model was compiled with a categorical cross-entropy loss function and the Adam optimizer with the learning rate of 0.001. Training was performed for 10 epochs with a batch size of 64.

In order to speed up training, I used a GPU (Nvidia T4) provided by Google Cloud.

Over the course of training, the model learned to predict the next character in a sequence of characters. The loss decreased with each epoch, indicating that the model was learning the patterns present in the training data. Epoch by epoch, the model 'learned' to use the concepts of common syllables, words, punctuation, line breaks, and finally sentence structures to some degree, producing more and more coherent text.

After 10 epochs, the model started to converge and the loss stabilized. The trained model was saved for future use in generating text.

2.7 Text Generation

To generate text, I provided the model with a seed sequence (a 40-character snippet from the corpus) and ask it to predict the next character. I then appended that predicted character to the seed and continued the process iteratively to generate 200 characters of text.

In order to control the creativity and coherence of the generated text, I introduced a **diversity parameter**. This parameter adjusts the probability distribution over characters before sampling. A lower diversity parameter leads to choosing the most probable character more frequently, resulting in more deterministic output. A higher diversity parameter allows the model to pick less likely characters.

I used three diversity levels to generate text samples: *low* (0.1), *medium* (0.5), and *high* (1.0). The expected outcome was that the generated text would be more predictable and coherent at low diversity, more creative at medium diversity, and more random at high diversity.

3 Analysis and Results

The LSTM model was successfully trained on the combined text of Jane Austen's novels. The training process involved creating overlapping sequences of 40 characters and their corresponding next characters. The model architecture consisted of two LSTM layers with 128 units each (64 units in each direction). The model was trained for 10 epochs using the Adam optimizer and a batch size of 64.

I generated text samples using the trained model with varying diversity parameters (0.1, 0.5, 1.0) and a seed text from Jane Austen's letters.

The complete quote used as the seed text was:

> *I do not want people to be very agreeable, as it saves me the trouble of liking them a great deal.*

I used the first 40 characters of this quote as the seed text for text generation:

```
I do not want people to be very agreeabl
```

200 characters generated with diversity = 0.1:

```
I do not want people to be very agreeable to her at the sister of the
stay of the subject of the same sister of the sister of the sister
of the sister and some thing the consideration of the sister of the
sister of her sister and happy and
```

---

200 characters generated with diversity = 0.5:

```
I do not want people to be very agreeable and her seens of the daughter.
"you will not speak that i have not all for the family, and at the
```

disappointed standing, and some months, and he had been considered and the telling her home, i have

---

200 characters generated with diversity = 1.0:

I do not want people to be very agreeable **brother passary to party, emma could do the visits afterwards and situation to me to like what say. very ascond, which can be cany acquainted super he had he continued her disappointed, which hobou**

The results confirm the expected behavior of the diversity parameter. At low diversity (0.1), the generated text looks more predictable but repetitive. Semantically they're all quite nonsensical, but here the generated words are coherent and are proper English words. At medium diversity (0.5), the text becomes more 'interesting' – the words are still correct, but the generated text is more 'creative' and less repetitive. At high diversity (1.0), the text becomes more random and less coherent, sometimes producing nonsensical words and sudden changes in context.

4 Conclusion

All in all, I successfully trained a character-based text generation model on Jane Austen's novels. The LSTM model learned to predict the next character in a sequence of characters and generated text samples based on a seed text. By adjusting the diversity parameter, I could control the creativity and coherence of the generated text.

While the generated text can somewhat mimic the style and character distribution of the source material, it still lacks a coherent narrative. Adjusting diversity parameters helps in controlling the trade-off between creativity and stability. With more data, improved architectures, or moving to word-level prediction (discussed in "Limitations and Future Work" section below), it should be possible to improve the quality of the generated text.

Speaking of challenges, personally the process of training the model and generating text was quite straightforward. One challenge I could mention is the training time. Training an LSTM on over 600 thousand sequences takes considerable computational resources. I used a Google Cloud instance with a Nvidia T4 GPU to speed up training. This allowed me to increase the batch size to 64 (although I could have gone higher) and process roughly 1 epoch per minute.

4.1 Limitations and Future Work

The current model has several limitations:

- **Poor semantic understanding:** The model's output lacks coherent

semantic meaning. It can generate text that looks similar to the training data but often lacks logical structure.

- **Short context window:** The model only considers the previous 40 tokens (characters). This really limits the context the model can use to generate the next token.
- **Character-level generation:** Character-level tokenization was used. While simple to implement, this further reduces the model's view of the broader context and the model quickly forgets long-range dependencies.

There are several ways to improve the model and experiment further:

- **Fine-tuning:** I didn't perform extensive hyperparameter tuning. Experimenting with different LSTM units, number of layers, dropout rates, etc., could improve the quality of the generated text.
- **Larger datasets:** Training on more extensive corpora could expose the model to a broader range of language patterns and improve the quality of generated text.
- **Larger token sizes:** Instead of character-based generation, either word-level or subword tokenization (e.g. Byte Pair Encoding) would enlarge the context window and could improve coherence. Additionally, a larger context window could be used.
- **Advanced model architectures:** Modern transformer-based models have shown superior performance in language modeling tasks. These models could be explored for text generation tasks.

5 Source Code

The source code used in this solution is available in the Appendix 1.

6 Appendix 1: Source Code

Source code used in this solution is presented below. The code is written in Python 3.11 and uses the following libraries: `nltk`, `numpy`, `torch`, `scikit-learn`.

```
import argparse
import logging
import pickle
from pathlib import Path

import nltk
import numpy as np
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, Dataset

nltk.download("gutenberg")
from nltk.corpus import gutenberg


logging.basicConfig(level=logging.INFO)

# constants
TORCH_DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
INPUT_DIR = Path("data/input")
OUTPUT_DIR = Path("data/output")
PLOT_DIR = Path("data/plots")
TRAINING_TEXTS = [
    "austen-emma.txt",
    "austen-persuasion.txt",
    "austen-sense.txt",
]


parser = argparse.ArgumentParser(
    description="Text Generation using LSTM",
)
parser.add_argument(
    "--mode",
    choices=["train", "generate"],
    required=True,
    help="Mode of operation",
)
parser.add_argument(
    "--sequence_length",
    type=int,
    default=40,
    help="Length of the character sequence for training",
)
```

```
parser.add_argument(
    "--step",
    type=int,
    default=3,
    help="Step size for creating sequences",
)
parser.add_argument(
    "--batch_size",
    type=int,
    default=64,
    help="Batch size for training",
)
parser.add_argument(
    "--tokens_to_generate",
    type=int,
    default=200,
    help="Number of tokens to generate",
)


class TextDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y

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

    def __getitem__(self, idx):
        return torch.tensor(self.X[idx], dtype=torch.float32),
            torch.tensor(
            self.y[idx], dtype=torch.float32
        )


class TextGenerator(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.input_size = input_size
        self.output_size = output_size

        self.lstm = nn.LSTM(
            input_size,
            hidden_size=64,
            num_layers=2,
            dropout=0.2,
            bidirectional=True,
            batch_first=True,
        )
        self.fc = nn.Linear(128, output_size)
```

```
    def forward(self, x, hidden=None):
        out, hidden = self.lstm(x, hidden)
        out = out[:, -1, :]
        out = self.fc(out)
        return out, hidden


def train_model(
    sequence_length: int,
    step: int,
    batch_size: int,
):
    logging.info("Training the model with the following parameters:")
    logging.info(f"Sequence length: {sequence_length}")
    logging.info(f"Step size: {step}")

    text = ""
    for filename in TRAINING_TEXTS:
        text += gutenberg.raw(filename) + "\n\n\n"

    # convert to lowercase
    text = text.lower()

    chars = sorted(set(text))
    logging.info(f"Number of characters: {len(text)}")
    logging.info(f"Number of unique characters: {len(chars)}")

    char_to_idx = {c: i for i, c in enumerate(chars)}
    idx_to_char = {i: c for c, i in char_to_idx.items()}

    sequences = []
    next_chars = []
    for i in range(0, len(text) - sequence_length, step):
        sequences.append(text[i : i + sequence_length])
        next_chars.append(text[i + sequence_length])
    logging.info(f"Number of sequences: {len(sequences)}")

    # transform inputs
    X = np.zeros((len(sequences), sequence_length, len(chars)),
         dtype=np.float32)
    y = np.zeros((len(sequences), len(chars)), dtype=np.float32)
    for i, seq in enumerate(sequences):
        for t, char in enumerate(seq):
            X[i, t, char_to_idx[char]] = 1.0
        y[i, char_to_idx[next_chars[i]]] = 1.0

    dataset = TextDataset(X, y)
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
    model = TextGenerator(
        input_size=len(chars),
        output_size=len(chars),
    )
    model.to(TORCH_DEVICE)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.002)
    criterion = nn.CrossEntropyLoss()

    logging.info("Training the model...")
    for epoch in range(10):
        model.train()

        for batch_x, batch_y in dataloader:
            batch_x = batch_x.to(TORCH_DEVICE)
            batch_y = batch_y.to(TORCH_DEVICE)
            optimizer.zero_grad()
            y_pred, _ = model(batch_x)
            loss = criterion(y_pred, batch_y)
            loss.backward()
            optimizer.step()

        model.eval()
        total_loss = 0
        for batch_x, batch_y in dataloader:
            batch_x = batch_x.to(TORCH_DEVICE)
            batch_y = batch_y.to(TORCH_DEVICE)
            y_pred, _ = model(batch_x)
            loss = criterion(y_pred, batch_y)
            total_loss += loss.item()

        logging.info(f"Epoch {epoch + 1}, Loss: {total_loss / (len(X) /
            batch_size)}")
    logging.info("Training complete.")

    torch.save(model, OUTPUT_DIR / "model.pth")
    logging.info(f"Model saved to {OUTPUT_DIR / 'model.pth'}")

    with open(OUTPUT_DIR / "char_maps.pkl", "wb") as f:
        pickle.dump((char_to_idx, idx_to_char), f)
    logging.info(f"Character maps saved to {OUTPUT_DIR /
        'char_maps.pkl'}")


def sample(preds, diversity):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype("float64")
    preds = np.log(preds + 1e-8) / diversity
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
```

```
    return np.argmax(probas)


def generate_text(
    model: torch.nn.Module,
    char_to_idx: dict,
    idx_to_char: dict,
    seed_text: str,
    diversity: float,
    tokens_to_generate: int,
) -> str:
    generated = ""
    sentence = seed_text[:40].lower() # TODO use parameter from the model

    for _ in range(tokens_to_generate):
        x = torch.zeros(1, len(sentence), len(char_to_idx))
        for t, char in enumerate(sentence):
            x[0, t, char_to_idx[char]] = 1.0
        x = x.to(TORCH_DEVICE)

        y_pred, _ = model(x)
        preds = torch.nn.functional.softmax(y_pred.cpu(),
            dim=1).detach().numpy()[0]
        next_index = sample(preds, diversity)
        next_char = idx_to_char[next_index]

        generated += next_char
        sentence = sentence[1:] + next_char

    return generated


if __name__ == "__main__":
    args = parser.parse_args()

    # create directories if they don't exist
    INPUT_DIR.mkdir(parents=True, exist_ok=True)
    OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
    PLOT_DIR.mkdir(parents=True, exist_ok=True)

    if args.mode == "train":
        logging.info("Training mode selected.")
        train_model(
            sequence_length=args.sequence_length,
            step=args.step,
            batch_size=args.batch_size,
        )

    elif args.mode == "generate":
        logging.info("Generate mode selected.")
```

```
filenames = list(INPUT_DIR.glob("*.txt"))
# validate input files
assert len(filenames) > 0, "No input files provided."

model = torch.load(
    OUTPUT_DIR / "model.pth",
    weights_only=False,
    map_location=torch.device(TORCH_DEVICE),
)
model.to(TORCH_DEVICE)
with open(OUTPUT_DIR / "char_maps.pkl", "rb") as f:
    char_to_idx, idx_to_char = pickle.load(f)

for filename in filenames:
    logging.info(f"Processing {filename}")
    with open(filename, "r") as f:
        seed_text = f.read()

    for diversity in [0.1, 0.5, 1.0]:
        logging.info(f"Diversity: {diversity}")
        output = generate_text(
            model=model,
            char_to_idx=char_to_idx,
            idx_to_char=idx_to_char,
            seed_text=seed_text,
            diversity=diversity,
            tokens_to_generate=args.tokens_to_generate,
        )

        logging.info("Seed text:")
        logging.info(seed_text[:40])
        logging.info("Generated text:")
        logging.info(output)
        logging.info("")
```