Next-Word Prediction with Recurrent Neural Networks

Overview

The goal of the assignment is to develop a neural network capable of predicting the next word in a sequence. The model utilizes a Long Short-Term Memory (LSTM) network that is well suited for sequential data, along with an attention mechanism. The provided dataset is "The Adventures of Sherlock Holmes" by Arthur Conan Doyle, sourced from Project Gutenberg. The project encompasses data downloading, preprocessing, model implementation, training, and text generation. I have broken down the end to end pipeline into 3 scripts:

1. downloader.py: Handles the download of the text file.
2. data_processor.py: Manages text cleaning, tokenization, and preparation of word embeddings.
3. next-word-predictor.py: Contains the core logic for model training and text generation.

Additionally I have included demo.ipynb - A Jupyter Notebook for demonstrating trained model outputs and interactive next-word predictions.

Overall Summary of Approach

My approach involves a complete pipeline for building a next-word prediction model. First, the text file that serves as training corpus is downloaded.
The following steps are followed:

- The text undergoes extensive preprocessing, including cleaning, sentence splitting, and custom word-level tokenization.
- A vocabulary is constructed, and word embeddings are generated using a pre-trained Sentence Transformer model ("all-MiniLM-L6-v2"). These embeddings are crucial for representing words in a dense vector space, allowing the model to understand semantic relationships.
- The core of the model is a custom-built neural network ("NextWordPredictor" class) comprising an embedding layer, an LSTM layer, an attention layer, and a final fully connected output layer.
- The model is trained using “nn.CrossEntropyLoss” and “AdamW” optimizer. During training, both loss and accuracy are monitored.
- Finally, a text generation function is implemented, which takes a seed text and iteratively predicts subsequent words, also providing insights into the model's decision-making process by showing the top 5 alternative predictions at each step.

Core Tasks and Experimental Details

1. **DATASET**: The “downloader.py” script is responsible for fetching this text file from the provided URL. I’ve added robust error handling with retries and random User-Agent headers to ensure successful downloading. The script also checks if the file already exists to avoid unnecessary re-downloads, which streamlines the development process.

2. **DATA PREPROCESSING:** The “data_processor.py” script performs comprehensive data preprocessing. Additionally I store intermediate results of creating data splits and embeddings vocabulary in pickle file to avoid redundancy in generating this multiple times.

    a. Text Cleaning: The raw text content is read and cleaned to remove unwanted characters, special symbols, and excessive whitespace. This includes replacing specific characters like '£', '½', '&', and various non-ASCII characters with their text equivalents or removing them. I also stripped Project Gutenberg's header and footer.

    b. Sentence Segmentation: The cleaned text is split into individual sentences using regular expressions, looking for common punctuation marks ('.', '!', '?') followed by whitespace.

    c. Chunking: Sentences are grouped into "chunks" of a certain length (e.g., aiming for chunks longer than 150 characters). This helps in managing long sequence lengths for training and ensures that the model sees enough context and continuing text.

    d. Train-Validation-Test Split: The generated chunks are shuffled and then divided into training (70%), validation (15%), and test (15%) sets.

    e. Word Level Tokenization: I used a custom tokenizer “custom_tokenize_with_spaces” to break down each chunk into words and punctuation marks, preserving spaces as tokens. This was crucial as it allows the model to learn patterns around spacing, which can be important for natural language generation.

    f. Vocabulary Building: I built a vocabulary from the tokenized words across all splits. Words with a frequency count less than 3 are excluded to manage vocabulary size and filter out rare words. I also added special tokens like “<PAD>”, “<UNK>” to the vocabulary.

g. Word Embeddings: Initially I tried learning embeddings from scratch but this resulted in an extremely poor performing model. With limited training data I believe using existing trained embeddings will result in strong word representations. I used the “all-MiniLM-L6-v2” pre-trained Sentence Transformer model to generate dense vector embeddings for each word in the vocabulary. For special tokens I added specific initializations: zero vector for “<PAD>”, and random small vectors for the remaining. This approach leverages the semantic knowledge already encoded in the pre-trained model, leading to better performance than randomly initialized embeddings, especially with smaller datasets.

h. Sequence Generation: Finally, the tokenized word chunks are converted into sequences of numerical indices based on the "word_to_idx" mapping. I use these sequences, along with their corresponding target sequences (shifted by one word for next-word prediction) to create “torch.TensorDataset” and “DataLoader” objects for efficient batch processing during training.

3. **MODEL IMPLEMENTATION:** The "NextWordPredictor" class defines the neural network architecture:

a. Embedding Layer: This layer takes the input word IDs and converts them into their dense vector representations using the pre-trained “embeddings_matrix”. I also allow the embeddings to be fine tuned during training, adapting them to specific dataset and tasks.

b. LSTM Layer: A Long Short-Term Memory (LSTM) network has the ability to capture long range dependencies in sequential data, mitigating the vanishing gradient problem common in vanilla RNNs. I used 256 units and experimented with 1 and 2 LSTM layers.

c. Attention Layer: I chose additive attention mechanism because it is suitable for sequence-to-sequence tasks and allows the model to selectively focus on different parts of the input sequence when making predictions for each word.

d. Fully Connected Layer: In this layer I combined LSTM and attention output to the size of the vocabulary, producing logits for each possible next word. The input dimension is "hidden_dim * 2" due to the concatenation of "lstm_out" and "attended_output".

e. Dropout Layer: I also added a dropout layer with a probability of 0.3 to prevent overfitting.

4. **TRAINING:** For training I used AdamW as the optimizer and CrossEntropyLoss for training. I also used OneCycleLR learning rate scheduler to apply gradient clipping to stabilize training. The model is trained iteratively, evaluating performance on a validation set and calculating accuracy and perplexity metrics after each epoch.

5. **TEXT GENERATION FUNCTION:** The generate_text function processes an input seed text, then iteratively predicts the next word for a desired length. I have added temperature scaling to control prediction randomness and provide the top 5 most likely next words with probabilities at each step, demonstrating the model's decision process. Additionally, I added a heuristic to mitigate excessive word repetition, to get more diverse and coherent output.

Experiments

The development involved several iterations, each addressing specific challenges and aiming for improved performance and coherence:

Version 0 (Initial Attempt):

a. Tokenization: Used Byte Pair Encoding (BPE).
b. Embeddings: Relied on PyTorch's "nn.Embedding" to learn embeddings from scratch.
c. Observation: Achieved a very low training accuracy (15%). This indicated that BPE, might have been too fine-grained for this relatively small dataset, and learning embeddings from scratch required more data or different hyperparameters.

Version 1 (Improved Tokenization and Embeddings):

a. Tokenization: Switched to custom word-level tokenization, including spaces as tokens, which proved to be more effective for this dataset.
b. Embeddings: Utilized pre-trained "all-MiniLM-L6-v2" Sentence Transformer for word embeddings. This significantly boosted the semantic understanding of words.
c. Vocabulary Size: Reduced the vocabulary size by filtering words with low frequency.
d. Epochs: Trained for 100 epochs.
e. Performance: Achieved significantly better results with Training Accuracy: 93.6% and Test Accuracy: 77.9%. This showed that leveraging pre-trained embeddings and appropriate tokenization was critical.

Version 2 (Addressing Repetition and Coherence):

a. Repetition Handling: Introduced a simple mechanism to reduce repetitive word generation.

b. Temperature Scaling: Implemented temperature scaling in the softmax function during text generation to control the randomness and improve coherence.
c. Epochs: Increased training to 125 epochs.
d. Performance: Further improved, reaching Training Accuracy: 95.92% and Test Accuracy: 80.31%. Coherence of generated text was notably better compared to Version 1.

Version 3 (Regularization and Sampling Exploration):

a. Regularization: Explored regularization techniques and Top-K Sampling to handle overfitting and improve coherence
b. Performance: Achieved a slight improvement in accuracy with Training Accuracy: 96.55% and Test Accuracy: 81.33%. Coherence remained comparable to Version 2, indicating that the primary coherence improvements were from temperature scaling and repetition handling.

Version 4 (Reduced number LSTM layer): I reduced LSTM layers from 2 to 1 that made the overall model less complex helping in faster convergence while not compromising on accuracy and coherence.

Final Metrics and Examples (Based on Version 2)

	Accuracy	**Perplexity**	**Loss**
Train	94.97%	1.29	0.3075
Validation	78.88%	3.57	1.2737
Test	80.31%	3.34	

Examples of Generated Text

Here are examples of text generated by the model with different seed phrases. The "num_words" generated for the first four examples is 50, and for the last one, it's 10, with "top5" details provided.

Example 1:
Generating from: 'Boots which extended halfway'
Generated text: boots which extended halfway over all that press and held a pounds work. what took place my earth, said he and! a secret was in his

Example 2:
Generating from: 'I ordered him to pay'
Generated text: i ordered him to pay over fifty pounds 1000 work. he was on professional business in discovering half, for a week after with his dreams and land when

Example 3:
Generating from: 'I answered that it had'
Generated text: i answered that it had been conviction if she carried out from his death but taken up by history and never mind to get for his work in my

Example 4:
Generating from: 'remove crusted mud from it. Hence'
Generated text: remove mud from it. hence ? well, how took to doctors round it, and so! what do my business knew how give me to buy his mother!

Example 5:
Generating from: 'He never spoke of the'
Generated text: he never spoke of the manner before him of life
--- Prediction Details ---
Step 1:
Top 5 likely words: (1.0000), - (0.0000), silence (0.0000), unpleasant (0.0000), ! (0.0000)
Step 2:
Top 5 likely words: bridegroom (0.3350), manner (0.2153), words (0.1213), death (0.0664), silence (0.0493)
Step 3:
Top 5 likely words: (0.9917), , (0.0055), . (0.0016), death (0.0009), ! (0.0002)
Step 4:
Top 5 likely words: himself (0.6891), never (0.2624), of (0.0239), which (0.0107), before (0.0034)
Step 5:
Top 5 likely words: (0.9976), , (0.0023), ? (0.0000), ! (0.0000), . (0.0000)
Step 6:'
Top 5 likely words: never (0.4794), he (0.2339), himself (0.1464), it (0.0485), him (0.0427)
Step 7:
Top 5 likely words: (0.9796), . (0.0174), , (0.0019), ? (0.0012), ! (0.0000)
Step 8:
Top 5 likely words: of (0.6853), for (0.0783), also (0.0774), in (0.0452), never (0.0243)
Step 9:
Top 5 likely words: (1.0000), - (0.0000), ? (0.0000), , (0.0000), . (0.0000)
Step 10:
Top 5 likely words: life (0.9782), strange (0.0063), himself (0.0051), <UNK> (0.0025), course (0.0015)