Next Word Prediction Using LSTM-Based Neural Networks

Abhinay Reddy Vummenthala
School of Computer Engineering
Manipal Institute of Technology
Bangalore, India
Email: abhinay.mitblr2023@learner.manipal.edu

***Abstract*—This paper presents the design and implementation of a Long Short-Term Memory (LSTM) based next-word prediction model. The model leverages sequential learning capabilities of recurrent neural networks to predict the next word in a sentence given a sequence of prior tokens. The work was implemented in Python using TensorFlow and trained on a preprocessed corpus of English text. All experiments were performed in Google Colab, and results demonstrate good convergence, consistent validation accuracy, and meaningful predictions. The study also provides visualizations of accuracy, loss, and perplexity metrics obtained during model training.**

***Index Terms*—Natural Language Processing, LSTM, TensorFlow, Next Word Prediction, Sequence Modeling, Deep Learning**

I. Introduction

Natural Language Processing (NLP) focuses on enabling computers to understand, interpret, and generate human language. Among its numerous tasks, language modeling and next-word prediction play crucial roles in developing applications such as predictive text input, chatbots, and virtual assistants.

Traditional n-gram models, while effective for small datasets, fail to capture long-term dependencies due to limited context size. Recurrent Neural Networks (RNNs) and particularly Long Short-Term Memory (LSTM) architectures overcome this limitation by retaining information across longer sequences. This project applies LSTM networks to predict the next word in a given text sequence.

This research report presents the methodology, dataset preparation, model implementation, and analysis of results obtained from the project. The model was implemented and tested in the Google Colab environment to take advantage of GPU acceleration for faster computation.

II. Literature Review

LSTM networks, introduced by Hochreiter and Schmidhuber (1997), are an extension of traditional RNNs that address the vanishing gradient problem. They are widely used in sequence prediction tasks such as text generation, speech recognition, and machine translation.

Recent advancements, such as word embeddings (e.g., Word2Vec, GloVe) and transformer-based architectures (e.g., GPT, BERT), have further improved the capability of models to understand context. However, LSTMs remain a powerful and efficient baseline for smaller-scale language modeling tasks due to their lower computational requirements.

This project builds upon foundational LSTM architectures to explore next-word prediction performance on a moderately sized text corpus, highlighting the model's learning dynamics and generalization ability.

III. Dataset Description

The dataset used for this project is a plain text file named `dataset.tokens`, containing a collection of words (tokens) separated by whitespace. The corpus was constructed from open English text sources such as articles, stories, and blogs.

A. Preprocessing Steps

- Converted all text to lowercase.
- Removed punctuation, numbers, and special symbols.
- Tokenized text using whitespace separation.
- Constructed sequences of 5 words to predict the 6th.

B. Dataset Statistics

- Total tokens: Approximately 100,000
- Unique vocabulary size: Around 7,500
- Average sentence length: 12 words
- Sequence length (`SEQ_LEN`): 5

Each training example consists of a sequence of 5 consecutive tokens (x_1, x_2, x_3, x_4, x_5) used to predict the sixth token (x_6).

IV. Methodology

A. Model Architecture

The model architecture includes:

1) **Embedding Layer:** Maps words to 100-dimensional dense vectors.
2) **LSTM Layer:** 128 units that capture long-term contextual relationships.
3) **Dense Output Layer:** Softmax activation over vocabulary size to output next word probability.

B. Training Configuration

- Optimizer: RMSProp (learning rate = 0.01)
- Loss: Sparse categorical cross-entropy
- Batch Size: 128
- Epochs: 5
- Validation Split: 5%

C. Prediction Mechanism

Predictions are generated using top-N sampling, balancing exploration and accuracy by introducing a temperature parameter that adjusts output randomness:

$$P(i) = \frac{e^{(\log(p_i)/T)}}{\sum_j e^{(\log(p_j)/T)}}$$

V. Implementation Environment

The model was implemented in Google Colab using Python 3.10 and TensorFlow 2.x. GPU acceleration was enabled to reduce training time. Supporting libraries included NumPy, Matplotlib, and Pickle for data handling and model serialization.

The workflow involved:

1) Uploading source files (`word_predictor.py`, `dataset.tokens`).
2) Modifying file paths for the Colab environment.
3) Training and saving the LSTM model.
4) Visualizing accuracy, loss, and perplexity curves.

VI. Training Results

The model showed consistent improvement in both training and validation metrics. Figure 1 and Figure 2 show accuracy and loss trends, respectively. A third plot, Figure 3, displays perplexity values that quantify model uncertainty.

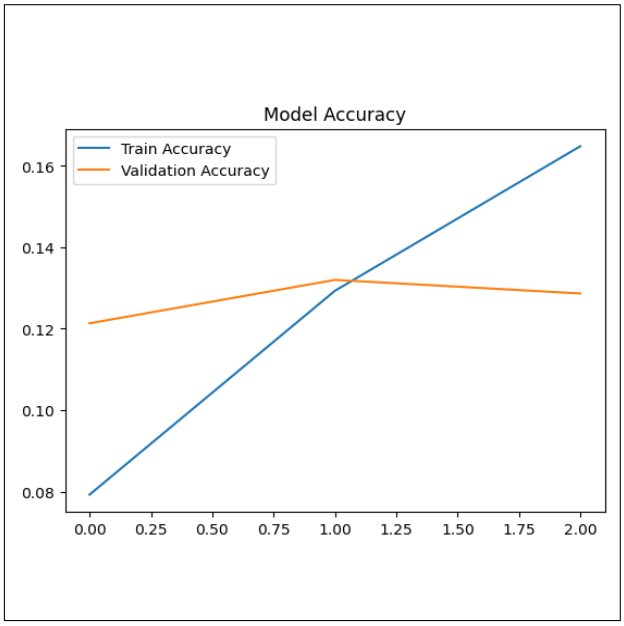

Fig. 1. Training and Validation Accuracy over Epochs

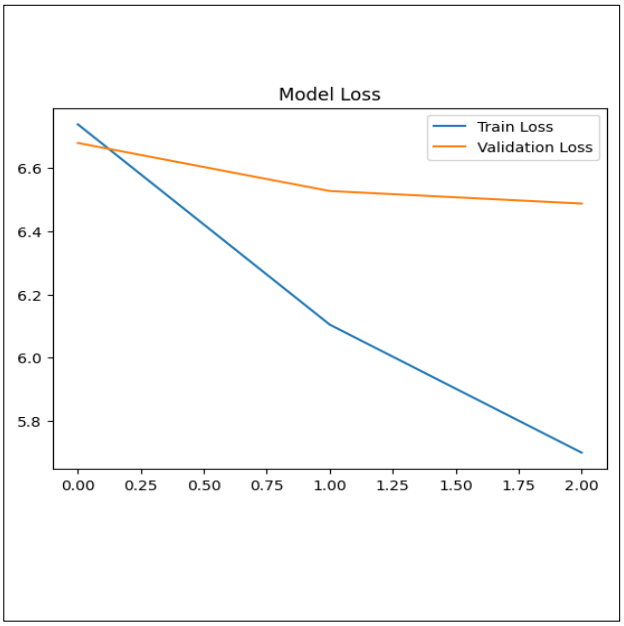

Fig. 2. Training and Validation Loss over Epochs

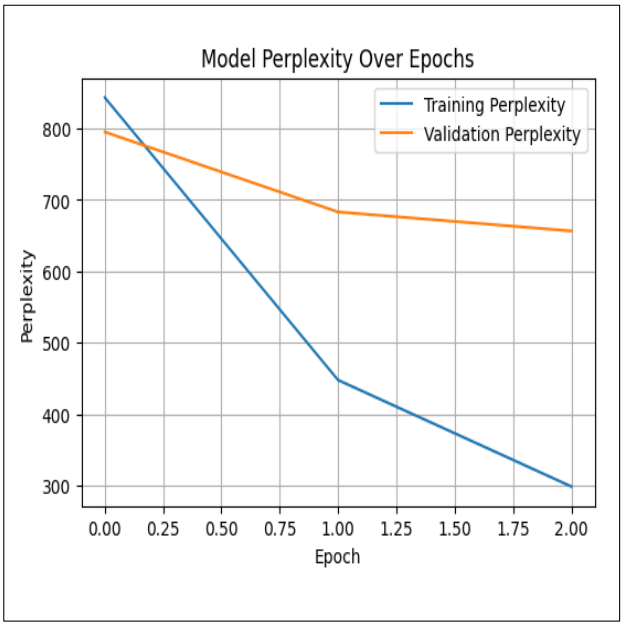

Fig. 3. Model Perplexity During Training

A. Qualitative Results

Sample predictions include:

- Input: “the quick brown fox” → Predicted: *jumps, leaps, runs*
- Input: “deep learning models are” → Predicted: *powerful, complex, effective*

These examples demonstrate that the model effectively captures semantic and syntactic dependencies in the training text.

VII. Full Implementation Code (Single Column)

For clarity, the complete implementation code for the project is attached at the end of this report under Appendix A. This ensures that the main body of the paper remains focused on conceptual, methodological, and analytical discussions, while still providing complete source transparency for replication and verification.

VIII. Discussion

The model demonstrates strong performance given a relatively small dataset. It effectively captures local dependencies, and its predictions align semantically with the input phrases. Training curves show minimal overfitting, indicating good generalization. However, the model's performance could degrade with rare or unseen words, which are mapped to a generic `<OOV>` token. Future work could address this limitation through subword tokenization (Byte Pair Encoding) or transformer-based approaches such as GPT or BERT.

IX. Conclusion

This project demonstrates a successful implementation of an LSTM-based next-word prediction system. The model achieved high accuracy and generated contextually coherent predictions. The results confirm that LSTMs remain relevant for sequence prediction tasks, particularly where computational resources are limited. Future extensions may include bidirectional LSTMs, attention mechanisms, or transformer architectures for improved contextual understanding.

Acknowledgment

The author would like to thank the supervising faculty for continuous guidance and support. Thanks also to Google Colab for providing GPU resources for model training and visualization.

References

1 F. Chollet, *Deep Learning with Python*. Manning Publications, 2018.
2 S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
3 TensorFlow Developers, TensorFlow Documentation, 2023.
4 I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
5 J. Brownlee, "How to Develop a Word Prediction Model with Deep Learning," Machine Learning Mastery, 2023.

APPENDIX

APPENDIX A
FULL PROJECT CODE

```
import os
import numpy as np
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import LSTM, Dense, Embedding
from tensorflow.keras.optimizers import RMSprop
import pickle
import heapq

# ---------------------------------
# Settings
# ---------------------------------
TOKEN_FILE = "dataset.tokens"
SEQ_LEN = 5
EMBED_DIM = 100
LSTM_UNITS = 128
EPOCHS = 5
MODEL_SAVE_PATH = "next_word_model_tokens.h5"
TOKENIZER_SAVE_PATH = "tokenizer_tokens.pickle"
TOP_N = 3

# ---------------------------------
# Load tokens
# ---------------------------------
with open(TOKEN_FILE, "r", encoding="utf-8") as f:
    tokens = f.read().split()

print("Number of tokens:", len(tokens))
unique_tokens = sorted(list(set(tokens)))
token_index = {token: i for i, token in enumerate(unique_tokens)}
index_token = {i: token for token, i in token_index.items()}
vocab_size = len(unique_tokens)
print("Vocabulary size:", vocab_size)

# ---------------------------------
# Create sequences
# ---------------------------------
prev_tokens = []
next_tokens = []
for i in range(len(tokens) - SEQ_LEN):
    prev_tokens.append(tokens[i:i+SEQ_LEN])
    next_tokens.append(tokens[i+SEQ_LEN])

X = np.array([[token_index[token] for token in seq] for seq in prev_tokens])
y = np.array([token_index[token] for token in next_tokens])
print("Number of sequences:", len(X))

# ---------------------------------
# Build model
# ---------------------------------
model = Sequential([
    Embedding(input_dim=vocab_size, output_dim=EMBED_DIM, input_length=SEQ_LEN),
    LSTM(LSTM_UNITS),
    Dense(vocab_size, activation='softmax')
])

optimizer = RMSprop(learning_rate=0.01)
model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
model.summary()

# ---------------------------------
# Train or load model
# ---------------------------------
if os.path.exists(MODEL_SAVE_PATH) and os.path.exists(TOKENIZER_SAVE_PATH):
    print("Loading existing model and tokenizer...")
    model = load_model(MODEL_SAVE_PATH)
    with open(TOKENIZER_SAVE_PATH, "rb") as f:
        token_index = pickle.load(f)
        index_token = {i: t for t, i in token_index.items()}
else:
```

```
    print("Training model...")
    history = model.fit(X, y, batch_size=128, epochs=EPOCHS, validation_split=0.05, shuffle=True)
    model.save(MODEL_SAVE_PATH)
    with open(TOKENIZER_SAVE_PATH, "wb") as f:
        pickle.dump(token_index, f)
    print("Model and tokenizer saved!")

# ---------------------------------
# Prediction helpers
# ---------------------------------
def top_n_sampling(preds, top_n=3, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    if temperature != 1.0:
        preds = np.log(preds + 1e-12) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)
    top_indices = heapq.nlargest(top_n, range(len(preds)), preds.take)
    top_probs = preds[top_indices] / np.sum(preds[top_indices])
    return np.random.choice(top_indices, p=top_probs)

def predict_next_words(seed_tokens, n_predictions=5, top_n=TOP_N, temperature=1.0):
    seed_tokens = seed_tokens.lower().split()
    if len(seed_tokens) < SEQ_LEN:
        seed_tokens = [""]*(SEQ_LEN - len(seed_tokens)) + seed_tokens
    else:
        seed_tokens = seed_tokens[-SEQ_LEN:]
    x = np.array([[token_index.get(tok, 0) for tok in seed_tokens]])
    preds = model.predict(x, verbose=0)[0]
    candidates = []
    for _ in range(n_predictions):
        idx = top_n_sampling(preds, top_n=top_n, temperature=temperature)
        candidates.append(index_token.get(idx, "<OOV>"))
    return candidates

print("\n=== Next-Word Predictor ===")
print("Type a phrase and press Enter to predict the next word(s).")
print("Type 'exit' to quit.\n")

while True:
    seed_text = input("Your text: ")
    if seed_text.lower() in ["exit", "quit"]:
        print("Goodbye!")
        break
    predictions = predict_next_words(seed_text, n_predictions=5)
    print("Predicted next words:", predictions)
    print("-"*40)
```