Report: Implementing an LLM Text Generator on the Persian Wikipedia Dataset

Introduction

In this project, we developed a text generator using a large language model (LLM) on the Persian Wikipedia dataset. The task involved implementing the model from scratch without utilizing pre-trained networks or fine-tuning existing models. This report details the steps taken, the challenges encountered, and how they were overcome.

Steps to Complete the Exercise

1. Dataset Creation

We created a text generation dataset from the Persian Wikipedia using the `torch.utils.data.dataset` class. The dataset was preprocessed to convert the text into a format suitable for training a neural network.

2. Model Implementation

The neural network for text generation was implemented from scratch. The architecture included an embedding layer, an LSTM layer, and a fully connected layer to output the predicted next word.

3. Training the Model

The model was trained on the created dataset with proper data handling, batching, and training loops. Training involved iterating over the dataset, calculating the loss, and updating the model's weights using backpropagation.

4. Model Evaluation

The model was evaluated using Perplexity, ROUGE, and other relevant evaluation metrics. Perplexity measures how well the model predicts the next word, while ROUGE scores measure the quality of the generated text compared to a reference text.

Challenges and Solutions

RAM Usage

Initially, the project faced significant RAM usage issues, especially when using the entire dataset. To address this, we took the following steps:

1. **Subset of Data**: We reduced the size of the dataset used for training. Starting with a subset of 1 million entries, we found that the memory usage was still too high. Therefore, we reduced the subset size to 1000 entries to test if the code worked.
2. **Gradual Increase**: After confirming that the code worked with 1000 entries, we gradually increased the subset size while monitoring RAM usage. This approach allowed us to find a balance between dataset size and memory usage.

Training Time

Training the model on the entire dataset took too long, which necessitated further adjustments:

1. **Reduced Epochs**: We reduced the number of epochs from a larger value to 5. This reduction allowed us to train the model within a reasonable time frame while still achieving meaningful results.
2. **Using GPU**: Moving the training process to a GPU significantly reduced the training time. The GPU's parallel processing capabilities enabled faster computations, improving the overall efficiency of the training process.

Additional Solutions

1. **Reduce Batch Size**: Lowering the batch size helped reduce memory consumption, although it might have increased training time.
2. **Truncate Dataset**: Used a smaller subset of the dataset for initial development and debugging, then scale up once the code is working.
3. **Sequence Length**: Reducing the sequence length also decreased memory usage.
4. **Model Complexity**: Simplify the model architecture (e.g., reducing the number of layers or hidden units).
5. **Gradient Accumulation**: This technique allowed us to simulate a larger batch size by accumulating gradients over several smaller batches before performing an optimization step.
6. **Checkpoints**: Save checkpoints and clear unnecessary variables from memory to manage resources efficiently.

Results

The final training and evaluation results are as follows:

Training Loss

- Epoch 1/5: Loss = 1.7003
- Epoch 2/5: Loss = 1.5726
- Epoch 3/5: Loss = 1.9304
- Epoch 4/5: Loss = 1.3571
- Epoch 5/5: Loss = 1.3596

Evaluation Metrics

- **Perplexity**: 4.2512
- **ROUGE Scores**:
    - ROUGE-1: Precision = 0.5, Recall = 0.3333, F1-Score = 0.4
    - ROUGE-L: Precision = 0.5, Recall = 0.3333, F1-Score = 0.4

Conclusion

This project successfully implemented a text generator using an LLM on the Persian Wikipedia dataset. Despite the challenges related to RAM usage and training time, we managed to overcome them by using a smaller subset of the data, reducing the number of epochs, and leveraging GPU for faster computations. Additionally, we employed several techniques to optimize memory usage and training efficiency, including

reducing batch size, truncating the dataset, adjusting sequence length, simplifying model complexity, using gradient accumulation, and saving checkpoints. The model demonstrated reasonable performance as indicated by the evaluation metrics, showing potential for further improvements with additional tuning and larger datasets.