Department of Computer Science
University of Cape Town, South Africa

NATURAL LANGUAGE PROCESSING: Language Modelling with Transformers

Prosper Arineitwe Asiimwe - ARNARI002

Claudious Nhemwa - NHMCLA003

August 2024

1 Overview

In this project, we focus on developing and training a Large Language Model (LLM) based on the transformer architecture, targeting the IsNdebele language (NR), which is classified as a low-resourced language in South Africa. We leveraged the NCHLT text corpora made available by the South African Centre for Digital Language Resources (SADiLaR) for our dataset.

2 Implementation Details

In this project, we implement character-level language modeling modeling. Our data was split into characters and to make the solution robust we applied tagging to show the end of sentence <EOS>, the start of sentence <SOS>, unknown <UNK> for words not in the vocabulary and padding <PAD>. Our model is based on the standard Transformer auto-regressive (decoder) language model architecture. The implementation was done in PyTorch, allowing for efficient experimentation. The model utilized various attention mechanisms, including standard multi-head attention, multi-query attention, and sparse attention. Positional encoding was added to provide information about character positions, which is crucial for understanding sequences. Key architectural hyperparameters included the hidden state size (d_model), number of layers, and number of attention heads. The initial configuration employed a hidden state size of 512, with four layers and eight attention heads. The architectural choices of a hidden state size of 512, 6 layers (for both encoder and decoder), and 8 attention heads were made to balance model complexity, training efficiency, and performance. These configurations are well-supported by evidence from multiple papers and online tutorials we discovered and align with established practices in designing Transformer models for language tasks. To enhance training efficiency and prevent overfitting, we incorporated a dropout strategy with a rate of 0.1 into our model. Mixed precision training was used to enhance performance on GPUs. Layer normalization was applied after the attention layers, as experiments showed this improved training stability. Weight tying between the embedding and output layers was also tested, which reduced model size without losing performance. A learning rate scheduler, specifically Cosine Annealing, was implemented to adjust the learning rate during training, helping the model converge more efficiently.

3 Training

To train our models we used google colab for computation. The colab offers a Tesla T4 with 16GB high-bandwidth memory (GDDR6). In our training, we used 10 training epochs with an early stopping mechanism with a patience of 3 epochs.

4 Discussion

4.1 Layer normalization and Weight tying

In our experiments, we assessed the impact of placing layer normalization before versus after the attention mechanism and the feed-forward network. Our observations indicated that positioning layer normalization post these elements led to a more rapid convergence of the model and enhanced the stability of the training process. Furthermore, upon introducing weight tying to our model, we noted an acceleration in the convergence rate.

4.2 Architecture Hyperparameter Tuning

The Transformer model used the following initial hyperparameters: Hidden State Size (d_model): 512, Number of Layers: 6, Number of Attention Heads: 8. These choices balance model capacity and efficiency. The PositionalEncoding module adds positional information, while MultiQueryAttention and SparseAttention provide alternative attention mechanisms. The TransformerModel class integrates embedding layers, positional encoding, and transformer blocks.

4.3 Optimization hyperparameter tuning

To determine the optimal batch size, we monitored GPU memory usage on Colab, testing batch sizes in increments of powers of two. We established that a batch size of 128, which utilized 13 GB of the available 16 GB of GPU memory, was optimal. Subsequently, we applied a similar approach to ascertain the ideal learning rate, using a vocabulary size of 100 and training at various learning rates. We found that a learning rate of $1e-2$ was the most suitable given Colab's computational constraints. From our experiments, we observed that increasing the learning rate resulted in faster convergence of the model, whereas decreasing the rate either slowed down the training process or caused the model to become stuck.

4.4 Attention Investigations

We explore three attention mechanisms: multi-head attention (MHA), sparse attention (SA), and multi-query attention (MQA). Supplementary files include training logs for each type. Both SA and MQA triggered early stopping after 8 epochs, whereas MHA completed the full 10 epochs. Despite this, MHA had a quicker average batch processing time of 1594 ms, compared to SA's 1658 ms and MQA's 1642 ms. Given the number of epochs, MQA was the most time-efficient.

During training, we found that MQA consumed less GPU memory on Colab than the other two, suggesting the potential to increase the batch size for faster training. In initial epochs, MQA and SA started with high losses and bits-per-character (bpc), with MQA at a loss of 427.96 and bpc of 617.4182, and SA at a loss of 420.02 and bpc of 605.9644. In contrast, MHA began with much lower initial bpc and loss at 16.4 and 11.4, respectively.

5 Conclusion

In this paper, we implement and train a Large Language Model using IsiNdebele a low-resourced language in South Africa. We investigate various attention types and hyper-parameters.