****

**DLP Project**

**Name: -**

Building an LLM from scratch – AMD LLM

**Group Members: -**

* 21K-3365 – Maaz M Khan
* 21K-3459 – Dinesh Dhanji
* 21K-4582 – Aneeq Muneer

**Github Repository: -**

https://github.com/DineshDhanji/LLM-from-Scratch.git

**Objective: -**

The goal of this project is to gain a deep, hands-on understanding of GPT-2 by reproducing its architecture from scratch, based on an open-source implementation available online. The project involves re-implementing each module, including multi-head self-attention, feedforward networks, and positional embeddings, using PyTorch without relying on high-level libraries like Hugging Face Transformers. Through this educational replication, we aim to solidify our understanding of the architecture, training dynamics, and inference process of autoregressive language models.

**Problem Statement: -**

Large Language Models (LLMs) like GPT-2 have significantly advanced the field of natural language processing, but their internal workings often remain hidden behind high-level abstractions provided by modern frameworks. For students and developers aiming to build a deeper understanding of how transformer-based architectures function, reproducing these models from scratch serves as an invaluable learning experience. However, building such models manually involves understanding complex components such as self-attention, layer normalization, and token embeddings, which are often overlooked when using pre-built libraries.

**Methodology: -**

In this project, we aim to build and train a Generative Pretrained Transformer (GPT) model from scratch to perform autoregressive language modelling. The entire workflow consists of several interconnected components, including data preparation, model architecture, training, and evaluation, all of which work together to generate meaningful text based on the learned patterns from the training data.

**Data Preparation and Tokenization:**The first step in the process involves preparing the dataset and tokenizing the input text. A custom dataset class is implemented to handle tokenization, using the GPT-2 tokenizer to convert raw text into tokens. These tokens are then grouped into overlapping chunks using a sliding window approach, allowing the model to learn context by predicting the next token in a sequence. To ensure efficient data handling during training, the data is fed into the model through a PyTorch DataLoader, which batches, shuffles, and streams the data for model consumption. The dataset used for training comes from the TinyStories collection, which is loaded and processed into appropriate sequences to train the model.

The dataset used for this project is the [Tiny Stories dataset by Ronen Eldan](https://huggingface.co/datasets/roneneldan/TinyStories) available on hugging face. It is a clean, synthetic dataset of very short, child-like stories written in simple English. It is ideal for training and experimenting on small language models due to its limited vocabulary, simple syntax and grammar, and short context length which is why it was chosen.

**Model Architecture:**The model is built using a simplified version of the GPT architecture, following the standard transformer-based structure that underpins autoregressive language models. It consists of embedding layers, multi-head self-attention layers, and feedforward networks arranged in transformer blocks. Each transformer block is equipped with residual connections, layer normalization, and dropout to ensure stable training and regularization. The model’s architecture is similar to GPT-2 (124M), with the primary difference being the reduced context length (256 tokens instead of 1024) due to resource constraints. The final output layer projects the embeddings onto the vocabulary space, producing logits that represent the model’s prediction for the next token in the sequence.

The model architecture in a better visual format is as follows:

* Transformer x 12
  + LayerNorm
  + Multi-Head Attention
    - 12 heads × 64 dimensions = 768 output
    - Uses causal masking to prevent attending to future tokens
  + Dropout (0.1)
  + Residual Connection
  + LayerNorm
  + Feedforward Network
    - Linear(768 → 3072) → GELU → Linear(3072 → 768)
  + Dropout (0.1)
  + Residual Connection
* LayerNorm
* Linear(768 → 50257) (Output logits)

**Model Training:**The training process involves feeding tokenized input data into the model and calculating the loss between the predicted token probabilities and the actual target tokens. The loss is computed using cross-entropy, and the model weights are updated through backpropagation. This process is repeated across multiple epochs to gradually minimize the loss and improve the model's performance. The optimizer is responsible for updating the weights, and the model's performance is evaluated at regular intervals using the validation set to ensure that the model generalizes well to unseen data.

During training, text generation is performed at regular intervals to evaluate the model's ability to generate coherent sequences based on the learned patterns. This text generation uses techniques like top-k sampling and temperature scaling to control the randomness of the output, producing more diverse or deterministic outputs based on the configuration.

**Model Evaluation:**Evaluation is conducted by comparing the model's performance on both the training and validation sets. Key metrics such as training loss, validation loss, and the total number of tokens processed are tracked to monitor the model’s progress. After training, the model is saved to a file for future use or evaluation. This allows for easy reloading of the model to generate additional text or fine-tune it on new data.

**Visualization and Analysis:**The training process is monitored visually by plotting the loss curves over time. This helps in understanding how well the model is learning, and if overfitting or underfitting is occurring. Training time is also recorded, providing insights into the efficiency of the training process.

**Text Generation and Fine-Tuning:**After training, the model is capable of generating text sequences based on a given prompt. Fine-tuning can be performed if further improvements are needed, either by training the model on additional data or modifying the existing hyperparameters. The generated text provides an indication of how well the model has learned the structure and nuances of language.

**Results: -**

The performance of the model was evaluated throughout the training process by monitoring both the training and validation loss, as well as periodically generating text samples to qualitatively assess the model’s ability to learn meaningful language patterns.

**Training & Validation loss: -**

The figure below illustrates how the validation loss steadily decreased over the observed training epochs, suggesting that the model was effectively learning from the data. The training loss curve shows early signs of convergence, reflecting the model's capacity to optimize token predictions even within the limited training window.

A screenshot of a computer

AI-generated content may be incorrect.

**Sample Text Generation: -**

After the model completed training, it was evaluated by generating text samples from different prompt inputs. The outputs showed a notable level of fluency, coherence, and context awareness, consistent with the goals of autoregressive language modelling.

A black screen with white text

AI-generated content may be incorrect.

**Reference: -**

This project is based on the open-source repository [LLMs-from-scratch](https://github.com/rasbt/LLMs-from-scratch.git) by Sebastian Raschka. The repository serves as the official codebase for the book Build a Large Language Model (From Scratch), which provides a comprehensive, step-by-step guide to developing a GPT-like large language model using PyTorch. The project encompasses various aspects of LLM development, including data preprocessing, model architecture, pretraining, and fine-tuning, offering both theoretical insights and practical implementations.