Relatório Final: Implementando o Llama

Matheus Palheta

[1]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Av. General Rodrigo Octávio, 6200 – Coroado I – 69080-900 – Manaus – AM – Brasil

matheus.palheta@icomp.ufam.edu.br

1. Visão Geral

Este documento fornece uma visão geral da implementação de um LLaMA (Large Language Model) em PyTorch. A implementação foca no treinamento de um modelo transformer usando um dataset que contém textos de Shakespeare. Inclui o download e a pré-processamento do dataset, definição da arquitetura do modelo, funções de treinamento, validação e teste. Na seção de referências, coloquei todos os materiais que utilizei para realizar este trabalho.

2. Decisões de projeto

Framework: O projeto utiliza PyTorch para a implementação do modelo devido à sua flexibilidade e amplo suporte.

Dataset: O dataset tinyshakespare é utilizado por ser pequeno o suficiente para uma demonstração, mas complexo o bastante para mostrar as capacidades do modelo. (O motivo principal é porque o professor mandou)

SentencePiece: SentencePiece é usado para tokenização, ajudando a lidar com unidades subword de forma eficaz. O SentencePiece funciona da seguinte maneira:

- *Treinamento do Tokenizador*: SentencePiece treina um modelo de tokenização em um corpus de texto. Durante o treinamento, ele aprende a dividir as palavras em subunidades (subwords) com base na frequência e probabilidade de ocorrência no corpus.
- *Unidades Subword*: Ao invés de dividir o texto em palavras completas, SentencePiece divide o texto em subwords, que podem ser prefixos, sufixos ou partes intermediárias de palavras. Isso permite que o modelo lide com palavras raras e morfologicamente ricas de maneira mais eficiente.
- *Vocabulários Fixos*: Gera um vocabulário fixo de subwords, permitindo que palavras desconhecidas ou novas sejam representadas de maneira consistente usando as subunidades aprendidas.
- *Tokenização e Detokenização*: Após o treinamento, SentencePiece pode tokenizar (encode) e detokenizar (decode) o texto de e para a sequência de subwords.

Seleção de Dispositivo: O código detecta e usa automaticamente o hardware disponível (CUDA, MPS ou CPU) para computação.

Divisão da Arquitetura em Classes: Uma decisão importante do projeto foi dividir os componentes da arquitetura do LLaMA em classes. Os motivos para essa decisão foram: , .

- *Modularidade*: Cada componente da arquitetura é encapsulado em uma classe separada, tornando o código mais organizado e fácil de entender.

- *Reutilização*: Componentes como camadas de atenção ou feed-forward podem ser reutilizados em diferentes partes do modelo ou em outros projetos.
- *Facilidade de Manutenção*: Alterações ou melhorias em um componente específico podem ser feitas de forma isolada, sem afetar o restante do código.
- *Teste e Depuração*: Testar e depurar componentes individuais se torna mais fácil, permitindo identificar e resolver problemas de maneira mais eficiente.

3. Arquitetura e Hiperparâmetros

A arquitetura do modelo de linguagem LLAMA, conforme ilustrada em 1, é uma variante da arquitetura Transformer com várias modificações e otimizações. A seguir, detalho cada componente da arquitetura.

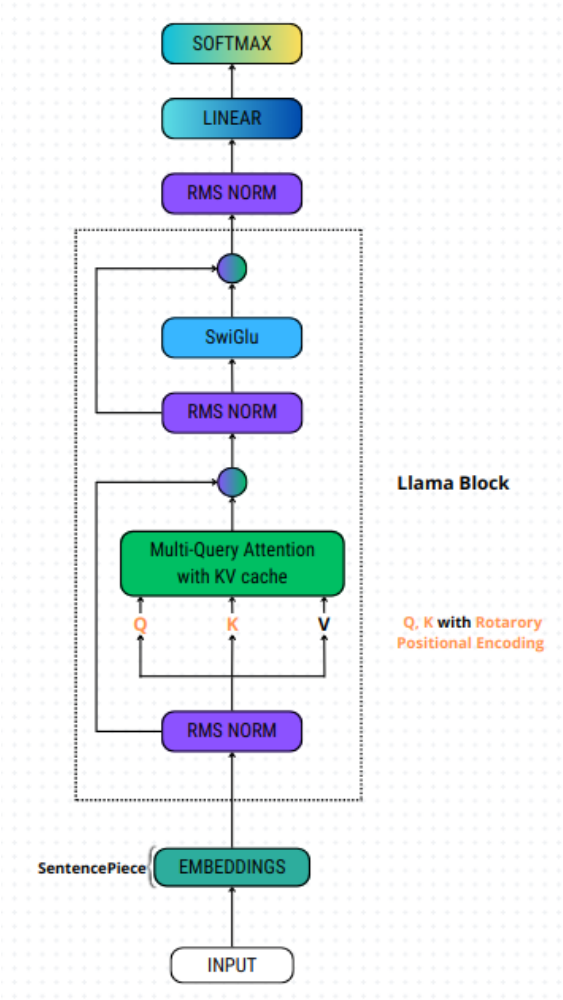

Figura 1. Arquitetura do Llama

Input: O ponto de entrada do modelo onde o texto original é fornecido. Recebe a sequência de texto que será processada pelo modelo.

Embeddings: Um processo de transformação onde cada token de entrada é convertido em uma representação vetorial densa. Usa a técnica de SentencePiece para dividir o texto em subpalavras e mapear estas subpalavras para vetores de embeddings.

Rotary Positional Encoding: Uma técnica de codificação posicional usada para incorporar informações sobre a posição dos tokens na sequência. Aplica a codificação rotativa às matrizes de consulta (Q) e chave (K) para preservar a ordem dos tokens na sequência.

RMS Norm (Root Mean Square Layer Normalization): Uma técnica de normalização que utiliza a média quadrática dos valores. Normaliza as ativações ao longo das camadas para estabilizar o treinamento e melhorar a performance.

Multi-Query Attention com KV Cache:Uma variante da atenção multi-cabeça onde as chaves (K) e os valores (V) são armazenados em cache para eficiência. Calcula a atenção para cada token da sequência de entrada, permitindo ao modelo focar em diferentes partes da sequência para cada token. A utilização do cache otimiza a reutilização de chaves e valores, melhorando a eficiência.

SwiGLU (Switching Gated Linear Units): Uma função de ativação composta por duas unidades lineares e uma operação de gating. Introduz não-linearidades no modelo, permitindo que ele capture representações mais complexas dos dados de entrada.

Llama Block: A unidade de construção principal do modelo LLAMA, que consiste em várias camadas de RMS Norm, Multi-Query Attention, e SwiGLU. Funcionalidade: Repete a estrutura interna para construir a profundidade do modelo. Cada bloco processa a entrada e passa sua saída para o próximo bloco na pilha.

Linear: Uma camada linear que transforma a saída do último bloco LLAMA. Reduz a dimensionalidade da saída do modelo para a dimensão do vocabulário.

Softmax Uma função de ativação que normaliza a saída da camada linear em probabilidades. Produz as probabilidades de cada token do vocabulário ser a próxima palavra na sequência.

A seguir, apresento os hiperparâmetros utilizados bem como a decisão para seus valores:

- **VOCAB_SIZE = 130**: Define o número total de tokens únicos que o modelo pode reconhecer e utilizar. Um vocabulário menor pode acelerar o treinamento, mas pode não capturar toda a riqueza do texto.
- **BATCH_SIZE = 32**: Especifica o número de exemplos de treinamento a serem processados em conjunto antes de atualizar os pesos do modelo. Um tamanho de lote maior pode levar a uma convergência mais estável, mas requer mais memória.
- **CONTEXT_WINDOW = 16**: O número de tokens de entrada que o modelo considera ao fazer previsões. Uma janela de contexto maior permite que o modelo capture dependências de longo alcance no texto.
- **EPOCHS = 1000**: O número de vezes que o conjunto de treinamento completo é passado pelo modelo. Mais épocas podem levar a um melhor aprendizado, mas também aumentam o risco de overfitting.
- **DIM = 128**: Especifica a dimensionalidade dos vetores de embedding e das camadas internas do modelo. Maior dimensionalidade permite que o modelo capture mais informações, mas também aumenta a complexidade computacional.
- **LOG_INTERVAL = 10**: Determina o intervalo de épocas após o qual as métricas de validação e treino são calculadas e registradas. Registrar métricas ajuda a monitorar o progresso do treinamento e ajustar hiperparâmetros conforme necessário.

- **HEADS = 8**: Define o número de cabeças de atenção (attention heads) em cada camada de atenção multi-cabeça. Mais cabeças permitem que o modelo atenda a diferentes partes da entrada de forma independente.
- **LAYERS = 4**: Especifica o número de camadas no modelo. Camadas adicionais podem permitir que o modelo capture representações mais complexas, mas também aumentam o tempo de treinamento e a complexidade computacional.
- **OTIMIZADOR = Adam**: O otimizador Adam é usado para ajustar os pesos do modelo durante o treinamento. Adam combina as vantagens dos métodos de AdaGrad e RMSProp para fornecer uma otimização adaptativa eficiente. Ele utiliza estimativas de primeira e segunda ordem dos momentos do gradiente para ajustar a taxa de aprendizado de cada parâmetro.

4. Treino e validação

O dataset foi dividido em três partes: treino (70%), validação (15%) e teste (15(%). A fim de evitar o overfitting, em cada iteração crio um batch aleatório independente do anterior. a métrica utilizada para atualização dos pesos do modelo foi a média por entropia cruzada, mostrada na fórmula abaixo:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

onde:

- $\mathcal{L}$ é a perda média.
- N é o número de exemplos no lote.
- C é o número de classes.
- $y_{i,c}$ é o valor verdadeiro da classe c para i.
- $\hat{y}_{i,c}$ é a probabilidade predita pelo modelo para a classe c para i.

- **Perda de Validação (Validation Loss)**: 4.469351
- **Acurácia de Validação (Validation Accuracy)**: 0.055902
- **Perplexidade de Validação (Validation Perplexity)**: 87.316005

5. Teste e Comparação

O modelo utilizado para comparação foi o GPT-2 também treinado na base de dados tinyshakespeare. As métricas escolhidas para comparação foram acurácia e perplexidade.

A fórmula da acurácia é dada por:

$$\frac{\sum_{i=1}^{N} \mathbf{1}(\hat{y}_i = y_i)}{N}$$

onde:

- N é o número total de exemplos.
- $\hat{y}_i$ é a previsão do modelo para o exemplo i.
- y_i é o valor verdadeiro para o exemplo i.
- $\mathbf{1}(\hat{y}_i = y_i)$ é uma função indicadora que retorna 1 se $\hat{y}_i = y_i$ e 0 caso contrário.

A fórmula da perplexidade é dada por:

$$2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 P(y_i|x_i)}$$

onde:

- N é o número total de palavras na sequência.
- $P(y_i|x_i)$ é a probabilidade predita pelo modelo para a palavra y_i dada a entrada x_i.

- **Acurácia de Teste (Test Accuracy)**: 0.056173
- **Perplexidade de Teste (Test Perplexity)**: 87.698829

6. Melhorias Futuras

Nesta seção, seguem possíveis direções para melhorias futuras na arquitetura criada durante o trabalho. Com base em estratégias utilizadas por outros modelos de linguagem.

- **Aprimoramento das Técnicas de Atenção**: Implementar técnicas avançadas de atenção, como a Atenção Escalonável (Scalable Attention) ou a Atenção com Sparsidade (Sparse Attention), que são utilizadas em modelos como o GPT-3 e o GLaM. Essas técnicas podem reduzir os requisitos computacionais e melhorar a eficiência do modelo.
- **Melhoria na Normalização**: Explorar variantes de normalização, como a Normalização com Estatísticas Condicionais (Conditional Layer Normalization), para melhorar a estabilidade do treinamento e a performance do modelo em diferentes tarefas.
- **Aprimoramento das Técnicas de Geração de Texto**: Implementar técnicas de decodificação avançadas, como a amostragem de temperatura ajustável ou o beam search com reranking, para melhorar a coerência e a relevância das respostas geradas.
- **Exploração de Modelos Mixtos (Mixture of Experts)**: Inspirado pelo GLaM da Google, considerar a implementação de uma arquitetura de Mixture of Experts, onde diferentes "experts"(sub-modelos) são ativados para processar diferentes partes do input. Isso pode melhorar a eficiência computacional e permitir a especialização do modelo em diferentes subdomínios.
- **Melhoria na Codificação Posicional**: Explorar alternativas para a codificação posicional, como a Codificação Posicional Aprendida ou a Codificação Relativa, para permitir que o modelo capture melhor as dependências entre tokens em diferentes posições.
- **Aprimoramento da Robustez e Segurança**: Implementar técnicas para melhorar a robustez e segurança do modelo, como a detecção e mitigação de vieses, e a implementação de filtros para evitar a geração de conteúdo inapropriado ou prejudicial.

Referências

AI, M. (2023). Original code implementation of llama 1. https://github.com/meta-llama/codellama/tree/main/llama.

Dani, S. (2023). Implementation fine tuning on gpt 2. https://github.com/shreydan/shakespeareGPT.

Karpathy, A. (2015). Tiny shakespeare dataset. Used data.

Kitano, B. (2023). Llama from scratch. https://github.com/bkitano/llama-from-scratch/tree/main.

Lomsadze, S. (2023). Fine-tune the gpt-2 model on the works of shakespeare. https://salomelomsadze.medium.com/fine-tune-the-gpt-2-model-on-the-works-of-shakespeare-24b5e56dfe04.

Shazeer, N. (2020). Glu variants improve transformer. https://arxiv.org/abs/2002.05202.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. (2023). Roformer: Enhanced transformer with rotary position embedding. https://arxiv.org/abs/2104.09864.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023). Llama: Open and efficient foundation language models. *Arxiv*.

Zhang, B. (2023). Rmsnorm implementation. https://github.com/bzhangGo/rmsnorm.

Zhang, B. and Sennrich, R. (2019). Root mean square layer normalization. *CoRR*, abs/1910.07467.