Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Implementação da Arquitetura Vision Transformer por meio de um Framework de Alto Desempenho.

Carlos Henrique Pereira Abreu

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Prof. Dr. Talles Henrique De Medeiros

Março, 2023
João Monlevade–MG

Carlos Henrique Pereira Abreu

Implementação da Arquitetura Vision Transformer por meio de um Framework de Alto Desempenho.

Orientador: Prof. Dr. Talles Henrique De Medeiros

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Março de 2023

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Carlos Henrique Pereira Abreu

Implementação da Arquitetura Vision
Transformer por meio de um Framework de Alto
Desempenho

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de bacharel em Sistemas de Informação

Aprovada em 30 de março de 2023

Membros da banca

Dr. - Talles Henrique de Medeiros - Orientador - DECSI - Universidade Federal de Ouro Preto
Dr. - Darlan Nunes de Brito - DECSI - Universidade Federal de Ouro Preto
Dr. - Eduardo da Silva Ribeiro - DECSI - Universidade Federal de Ouro Preto

Talles Henrique de Medeiros, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 15/05/2023.

Documento assinado eletronicamente por **Talles Henrique de Medeiros**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 15/05/2023, às 16:01, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0525117** e o código CRC **6E00FB6F**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

Este trabalho é dedicado a todos aqueles que de alguma forma me impactaram, e por este motivo são responsáveis, mesmo que de forma mínima, pelo que sou hoje. Lembrem-se sempre de serem cordiais, ajudar, motivar, discordar e ensinar a todos que você toca. O mundo é um só e você pode fazer a diferença.

Agradecimentos

Agradeço primeiramente a minha família, que sempre lutou para proporcionar o melhor para mim. Lembro de todas os sacrifícios feitos por vocês e por isso sou muito grato.

Aos meus amigos além de agradecer por estarem ao meu lado nos melhores e piores momentos, quero propor um brinde a vida. Viver ao lado de vocês foi e continuará sendo uma das melhores coisas. Foram muitas lições aprendidas e ensinadas, a troca de conhecimento e de experiência com vocês contribuiu e ainda contribui em muito para o que sou hoje, seja de forma profissional ou pessoal.

Ao divino agradeço por todas as bençãos postas sobre mim e peço que continue me guiando e me levantando quando eu perder a fé no mundo ou em mim mesmo.

Agradeço a meu orientador Talles Medeiros, por me instruir e despertar ainda mais meu interesse pela área de inteligência artificial. Foram longos meses e conseguimos juntos desenvolver muito coisa.

A mim mesmo faço um agradecimento como motivo de comemoração. Continue sendo esta pessoa cativante e curiosa. O seu mundo já é lindo, mesmo que algumas pessoas não entendam como ele funciona.

"Change will come to those who have no fear."

— Lera Lynn,
in Song: The Only Thing Worth Fighting For.

Resumo

Os *Transformers* atualmente representam uma das classes mais poderosas de modelos de aprendizado de máquina por sua versatilidade e alto poder de processamento de dados sequenciais de forma paralela, sem a necessidade de processar uma entrada sequencial de dados. Desde o seu surgimento em 2017, a arquitetura Transformer tem sido amplamente utilizada em muitos problemas de processamento de linguagem natural (PLN), incluindo tradução automática, análise de sentimentos, geração e classificação de textos, e tem sido amplamente considerada como um dos modelos mais poderosos disponíveis. Neste trabalho, exploramos a construção da rede Transformer aplicada a problemas de visão computacional, utilizando o framework Joey. Para a implementação, novas camadas e funções auxiliares foram incorporadas ao framework. Os resultados produzidos pela rede, mesmo que inferiores em comparação com o framework PyTorch, representam uma evolução da gama de possíveis aplicações para o Joey.

Palavras-chaves: arquitetura Transformer, visão computacional, Devito framework, Joey, Python.

Abstract

Transformers currently represent one of the most powerful classes of machine learning models due to their versatility and high processing power for sequential data in parallel, without the need to process input data sequentially. Since their emergence in 2017, the Transformer architecture has been widely used in many natural language processing (NLP) tasks, including machine translation, sentiment analysis, text generation, and text classification, and has been widely considered as one of the most powerful models available. In this work, we explore the construction of the Transformer network applied to computer vision problems using the Joey framework. For implementation, new layers and auxiliary functions were incorporated into the framework. The results produced by the network, although inferior compared to the PyTorch framework, represent an evolution of the range of possible applications for Joey.

Key-words: Transformer architecture, computer vision, Devito framework, Joey, Python.

Lista de ilustrações

Lista de abreviaturas e siglas

HPC *High-performance Computing*

PDE Equação Diferencial Parcial

CPU Unidade Central de Processamento

GPU Unidade de Processamento Gráfico

LDE Linguagem de Domínio Especifico

IDE *Integrated Development Environment*

RNP Rede Neural Profunda

RNN Rede Neural Recorrente

CNN Rede Neural Convolucional

MPI *Message Passing Interface*

JIT *Just in Time*

MLP *Multilayer Perceptron*

LSTM *Long Short-Term Memory*

IET *Interactive expression tree*

DLE *Devito Loop Engine*

VGG *Very Deep Convolutional Network*

Lista de símbolos

γ	Letra grega minúscula gama
ϵ	Epsilon
β	Beta
$\in$	Pertence
$\forall$	Para todo

Sumário

1 Introdução

As redes neurais profundas (RNPs) são modelos de aprendizado de máquina que consistem em múltiplas camadas interconectadas de neurônios, que permitem o processamento de informações complexas e a realização de tarefas sofisticadas em diversas áreas, como visão computacional, reconhecimento de fala, processamento de linguagem natural, entre outras Goodfellow, Bengio e Courville (2016). Com o avanço das técnicas de treinamento e otimização de RNPs, surgiram arquiteturas cada vez mais sofisticadas, como as redes convolucionais (CNNs) LeCun et al. (1989) e as redes neurais recorrentes (RNNs) **??**), que permitiram melhorias significativas em diversas aplicações.

No entanto, um dos principais desafios enfrentados pelas RNPs de grande porte é o aumento do número de parâmetros, que pode aumentar significativamente os custos de processamento, tornando-as impraticáveis para muitas aplicações em larga escala Bengio, Simard e Frasconi (1994). A introdução da arquitetura Transformer Vaswani et al. (2017), trouxe uma nova abordagem destinada para o processamento de linguagem natural, sendo esta arquitetura um exemplo de RNP que pode ser particularmente afetada por esse problema de escalabilidade, devido ao seu alto número de parâmetros.

Desde a sua publicação, as redes Transformers se tornaram objeto de estudo e aprimoramento por diversos pesquisadores e empresas, produzindo notórios resultados como a rede GPT-3 Brown et al. (2020) e a rede BERT Devlin et al. (2018). Com a proposta de ser mais rápida e paralelizável que as redes neurais recorrentes, tranformers hoje configuram o estado da arte para problemas de processamento de linguagem natural.

Nesse contexto, a biblioteca Devito Kukreja et al. (2016) se destaca como uma ferramenta capaz de otimizar o desempenho computacional de operações matriciais e reduzir os custos de processamento de determinados problemas. O Devito é uma biblioteca Python de alto desempenho que permite a geração eficiente de códigos para a solução numérica de equações diferenciais, o que é especialmente útil para aplicações em física e engenharia. Com o suporte do Devito, é possível otimizar a execução de códigos que envolvem equações diferenciais parciais, reduzindo significativamente o tempo de processamento e tornando-as mais acessíveis para aplicações em larga escala.

Como resultado de iniciativas de estudo envolvendo a aplicação de um framework para computação de alta performance, foi desenvolvido o Joey Chatzitheoklitos (2020), um framework que combina a implementação das operações fornecidas pelo Devito com camadas projetadas para a reprodução de redes neurais convolucionais simples, como a LeNet LeCun et al. (1989).

Neste trabalho de conclusão de curso, discorreremos sobre a evolução das RNPs até

a chegada da arquitetura Transformer, elencando os desafios enfrentados pelo aumento do número de parâmetros e os problemas de custo computacional acerca de redes complexas e profundas. Em seguida, exploraremos como a biblioteca Devito pode auxiliar na redução desses custos e aprimorar as RNPs criadas, com destaque para a geração de código otimizado para solução de equações diferenciais. Por fim, apresentaremos exemplos de aplicações do Devito em RNPs de larga escala, como a classificação de imagens.

1.1 O problema de pesquisa

Analisando o atual framework Joey, que possui algumas camadas destinadas a construção de RNPs, já codificadas, e, sua capacidade de otimização através da compilação em tempo real e otimizada que o Devito oferece, existem abordagens e implementações não exploradas na ferramenta.

Espera-se que com a implementação de novas camadas e operações auxiliares, a arquitetura da rede Transformer possa ser reproduzida e os resultados de desempenho produzidos pela rede, comparados com outras implementações baseadas em diferentes frameworks.

1.2 Objetivos

O presente trabalho consiste na implementação das operações matriciais que viabilizarão a geração das camadas específicas para a construção da rede Transformer para problemas de visão composicional (ViT), a construção da rede em si, a submissão de uma versão atualizada do framework Joey, bem como uma análise comparativa de performance e tempo com a mesma rede construída utilizando o framework PyTorch. Com esta implementação finalizada, espera-se ganhos de desempenho de tempo computacional e consumo de recursos durante a fase de validação da rede.

Este trabalho possui aos seguintes objetivos específicos:

- Entender como o modelo Transformer aplicado a visão computacional funciona
- Identificar quais camadas e operações são necessárias para construção de um modelo classificador de imagens utilizando a arquitetura Transformer
- Desenvolver camadas faltantes para o framework Joey
- Implementar o modelo classificador de imagens baseado no framework Joey
- Validar as saídas do novo modelo implementado, comparando com o o framwork referência PyTorch

- Comparar o tempo de processamento para cada época de treinamento em cada uma das redes
- Comparar a performance de ambas as redes
- Submeter a versão atualizada do Joey para seu repositório oficial

1.3 Metodologia

Neste capítulo, serão apresentados os métodos e técnicas utilizados para a realização deste trabalho de conclusão de curso, que tem como objetivo de pesquisa verificar se a construção de uma rede neural Transformer em um framework focado em computação de alto desempenho, consegue entregar resultados de performance (tempo e consumo de recursos computacionais) melhores que o framework referência PyTorch Paszke et al. (2019).

1.3.1 Metodologia de pesquisa

O presente trabalho se caracteriza como um trabalho de desenvolvimento, levando em consideração a evolução das redes neurais, modelos, ferramentas e frameworks previamente desenvolvidos. As fontes bibliográficas foram obtidas por meio de livros, artigos científicos, dissertações e teses relacionadas ao tema do trabalho.

1.3.2 Metodologia de desenvolvimento

Foi adotada a metodologia de desenvolvimento de software em cascata para o aprimoramento do atual framework Joey, que consiste em uma abordagem sequencial para o desenvolvimento de software, com cada fase dependendo da fase anterior. Essa abordagem foi escolhida por ser mais adequada para o desenvolvimento de um framework robusto e completo.

1.3.3 Ferramentas utilizadas

Para o desenvolvimento da nova versão atualizada do framework Joey, foram utilizadas ferramentas de desenvolvimento integrado (IDE) PyCharm, focada na linguagem Python, GitHub para controle e versionamento do código, a plataforma do *Google Colaboratory (Colab)* que oferece um ambiente de execução de scripts para a linguagem Python, a biblioteca e plataforma *Weights and Bias*, que oferece um conjunto de ferramentas para monitorar o consumo de recursos das máquinas utilizadas na etapa de treinamento e validação, bem como a e elaboração de gráficos comparativos.

1.3.4 Processo de desenvolvimento

O processo de melhoria do framework atual foi dividido em fases, que incluíram verificação do seu atual funcionamento e limitações, estudo da arquitetura ser clonada, levantamento das camadas e operações a serem implementadas, desenvolvimento concomitantes com testes comparativos, operação a operação, camada a camada.

1.3.5 Limitações do desenvolvimento

As limitações do desenvolvimento incluem a disponibilidade limitada de recursos de hardware e software e o curto período de tempo disponível para o aprimoramento do framework, sendo este trabalho, destinado ao desenvolvimento de camadas e operações não existentes e adaptações nas camadas atuais, focadas na reprodução da arquitetura em estudo. Além disso, algumas funcionalidades poderão ser adicionadas em futuras versões do framework.

1.4 Organização do trabalho

O presente trabalho esta divido em 5 capítulos, sendo o Capítulo 1 a apresentação do tema, identificação do problema e objetivos da pesquisa de desenvolvimento. O Capítulo 2 apresenta a parte de revisão bibliográfica, abordando a evolução das redes neurais profundas até a concretização das redes transformes, bem como um detalhamento de como é o funcionamento do seu mecanismo principal, a *MultiHeadAttention*. O capítulo conta com dois exemplos das principais adaptações propostas à arquitetura Transformer original e como o modelo pode ser aplicado ao campo da visão computacional.

A introdução a área de computação de alto desempenho auxilia como instrumento introdutório ao framework/Linguagem de domínio específico (LDE) Devito, que busca implementar os conceitos da computação de alto desempenho (HPC) para a geração de códigos aprimorados em C, assim como arquitetura auxiliar na implementação de problemas que envolvem computação distribuída. O capítulo segue com a introdução ao Joey, framework derivado do Devito, focado na implementação de algumas camadas para a construção de redes neurais simples.

O capítulo 3 realiza primeiramente uma introdução aos experimentos iniciais feitos durante o processo de desenvolvimento, seguindo para o subcapítulo 3.0.3, onde a implementação das camadas faltantes é detalhada de forma lógica e sequencial para a construção da rede alvo.

Os resultados obtidos e testes aplicados são descritos no capítulo 4 e seguem para a conclusão do trabalho, considerações finais e propostas de trabalhos futuros, localizados no Capítulo 5.

2 Revisão bibliográfica

Neste capítulo é apresentado o levantamento bibliográfico utilizado neste estudo. Para contextualização do desenvolvimento do trabalho, será apresentado um breve histórico da evolução das RNPs, uma introdução sobre como a performance desses modelos pode ser afeta devido o tamanho da rede, conceitos de HPC, introdução ao framework Devito e seu derivado Joey.

2.1 Evolução das redes neurais

A evolução das redes neurais começou com as redes neurais *feedforward* simples, também chamadas de *multilayer perceptrons* (MLP), que foram seguidas pelas redes neurais profundas, com múltiplas camadas escondidas. Em seguida, vieram as redes convolucionais, CNNs, que foram projetadas especificamente para processar dados espaciais, como imagens. Essas redes neurais se tornaram populares para tarefas de classificação de imagem, segmentação de imagem e detecção de objetos. (GOODFELLOW; BENGIO; COURVILLE, 2016)

Em seguida, surgem as redes recorrentes, desenvolvidas para lidar com dados sequenciais, como séries temporais, texto e áudio. Essas redes neurais são capazes de capturar relações temporais entre dados sucessivos, tornando-as ideais para tarefas como previsão de séries temporais, análise de sentimentos e tradução automática. (GOODFELLOW; BENGIO; COURVILLE, 2016)

Finalmente, as redes de arquitetura Transformer Vaswani et al. (2017) surgiram como uma evolução das redes recorrentes. Elas foram projetadas especificamente para lidar com dados sequenciais de forma mais eficiente e escalável, permitindo o processamento paralelo de dados sequenciais. Além disso, a arquitetura permite a adição de camadas extras para aumentar a profundidade do modelo e aprimorar sua capacidade de aprendizado. Essas redes neurais são hoje muito populares para tarefas de processamento de linguagem natural, como tradução automática, análise de sentimentos, geração de texto e classificação de tópicos. (GOODFELLOW; BENGIO; COURVILLE, 2016)

2.1.1 Redes Convolucionais

As redes convolucionais, também conhecidas como Convolutional Neural Networks (CNNs), surgiram como uma evolução das redes neurais tradicionais, que eram utilizadas para tarefas de reconhecimento de padrões e classificação de dados. As CNNs foram desen-

volvidas especificamente para processamento de imagens, onde é necessária a identificação de padrões locais em uma imagem. (GOODFELLOW; BENGIO; COURVILLE, 2016)

O conceito de convolução já existia há muito tempo em processamento de sinais e imagem. A convolução é uma operação matemática descrita como um filtro, também chamado de *kernel*, que é aplicado a uma imagem. A operação de convolução é usada para extrair recursos locais, como bordas, cantos, texturas e outros detalhes, que são importantes para o reconhecimento de objetos em uma imagem. (GOODFELLOW; BENGIO; COURVILLE, 2016)

A ideia de usar convoluções em redes neurais para processamento de imagem foi proposta pela primeira vez na década de 1980, mas a grande revolução das CNNs ocorreu em 1998, quando Yann LeCun, junto com seus colegas, apresentou a rede neural LeNet-5 para reconhecimento de caracteres manuscritos. A LeNet-5 foi a primeira CNN a ser usada com sucesso em uma aplicação real de reconhecimento de padrões e, desde então, as CNNs tornaram-se a arquitetura mais comum para processamento de imagens. (LECUN et al., 1989)

Uma das principais vantagens das CNNs é a sua capacidade de aprendizado de recursos. As camadas convolucionais de uma CNN aprendem recursos locais, como bordas e cantos, de maneira automática a partir dos dados de treinamento. A combinação desses recursos em camadas posteriores permite que a rede aprenda representações cada vez mais abstratas das imagens, permitindo reconhecimento de objetos com alta precisão. (GOODFELLOW; BENGIO; COURVILLE, 2016)

Desde então, muitas arquiteturas de CNNs foram desenvolvidas, incluindo a famosa rede neural *VGG*, a *ResNet* e a *Inception*. As CNNs tornaram-se uma ferramenta fundamental em áreas como reconhecimento de imagem, visão computacional, processamento de linguagem natural, e muitas outras áreas de Inteligência Artificial.

2.1.2 Redes Recorrente

As Redes Neurais Recorrentes (RNNs) são uma evolução das redes neurais tradicionais que foram projetadas especificamente para lidar com dados sequenciais. Diferentemente das redes neurais tradicionais, que processam dados independentes entre si, as RNNs podem manter informações sobre o contexto de dados anteriores e usá-los para prever o próximo item na sequência. (GOODFELLOW; BENGIO; COURVILLE, 2016)

O conceito de redes neurais recorrentes foi introduzido pela primeira vez na década de 1980, mas foi em 1997 que Hochreiter e Schmidhuber apresentaram a arquitetura Long Short-Term Memory (LSTM), que se tornou a base para muitas outras redes neurais recorrentes modernas. (HOCHREITER; SCHMIDHUBER, 1997)

A ideia principal por trás das RNNs é que elas possuem um *"loop"* interno que

possibilita a transmissão de informações de uma unidade para a outra. Essa estrutura permite que a rede armazene informações sobre itens anteriores na sequência, o que é especialmente útil em tarefas de processamento de linguagem natural, onde a compreensão do significado de uma palavra pode depender das palavras que a precedem.

As RNNs são compostas por três tipos de camadas: entrada, oculta e saída A camada de entrada recebe a sequência de dados, e a camada oculta processa a sequência e mantém uma representação interna do estado da rede. A camada de saída produz as previsões.

A LSTM, em particular, é projetada para evitar o problema de desaparecimento do gradiente, que ocorre em redes neurais recorrentes tradicionais, onde o gradiente da função de perda diminui exponencialmente à medida que se retrocede na sequência, tornando o treinamento difícil. (GOODFELLOW; BENGIO; COURVILLE, 2016)

Desde então, muitas outras arquiteturas de redes neurais recorrentes foram desenvolvidas, como as redes neurais recorrentes bidirecionais (BiRNNs), que usam duas RNNs em direções opostas para capturar informações do passado e do futuro, e as redes neurais recorrentes de memória de atenção (RAM), que incorporam um mecanismo de atenção que permite que a rede se concentre em partes relevantes da sequência. (GOODFELLOW; BENGIO; COURVILLE, 2016)

As RNNs são amplamente utilizadas em aplicações de processamento de linguagem natural, como tradução automática, geração de texto e análise de sentimentos. Além disso, também são usadas em aplicações de visão computacional, processamento de áudio e outras áreas de inteligência artificial.

2.2 Transformers - Estado da arte

As redes de arquitetura Transformer são tipos de redes neurais capazes de processar dados sequenciais, como texto, áudio ou imagens em série. A arquitetura Transformer foi introduzida pela primeira vez em 2017 e tem sido amplamente utilizada em tarefas de processamento de linguagem natural, como tradução automática, análise de sentimentos, geração de texto e classificação de tópicos. (VASWANI et al., 2017)

A principal vantagem da arquitetura Transformer é sua capacidade de processar dados sequenciais de forma paralela, sem a necessidade de processar uma entrada sequencial passo a passo. Isso torna a arquitetura Transformer muito eficiente para treinar modelos de redes neurais em grandes conjuntos de dados.

Além disso, a arquitetura Transformer é escalável e permite a adição de camadas adicionais para aumentar a profundidade do modelo e aprimorar sua capacidade de aprendizado. Isso tem levado ao desenvolvimento de diversas derivações da arquitetura

Transformer, incluindo o BERT (Bidirectional Encoder Representations from Transformers) Devlin et al. (2018) e o GPT-3 (Generative Pretrained Transformer 3) Brown et al. (2020).

2.2.1 Arquitetura

A arquitetura Transformer original Vaswani et al. (2017), é construída por um bloco de codificadores (6 camadas de *encoder*) e outro bloco de decodificadores (este também com 6 camadas de *decorder*), como detalhado na figura 1. O modelo utiliza mecanismos de auto atenção para capturar as dependências de longo alcance em uma sequência de entrada.

A entrada do modelo é uma sequência de vetores de palavras (*embedding*) que são passados para uma série de camadas de codificação. Cada camada de codificação contém dois subcampos: um módulo de auto atenção e um módulo de *feed-forward*. O módulo de auto atenção calcula pesos de atenção para cada palavra em relação a todas as outras palavras da sequência, permitindo que o modelo dê mais importância às palavras relevantes para a tarefa em questão. O módulo de *feed-forward*, por sua vez, processa cada palavra individualmente. (VASWANI et al., 2017)

O modelo também usa um mecanismo de normalização residual e uma camada de saída linear após cada camada de codificação. O processo de codificação é realizado várias vezes para permitir que o modelo capture informações de diferentes granularidades.(VASWANI et al., 2017)

Depois de serem codificadas, as sequências de entrada são passadas para o decodificador, que é semelhante ao codificador, mas também incorpora informações da saída do modelo. O decodificador também utiliza mecanismos de auto atenção para gerar uma saída em cada etapa do processo.

2.2.1.1 Mecanismos de atenção

O mecanismo de atenção da arquitetura funciona de maneira semelhante à atenção humana. O objetivo é dar mais importância a algumas partes da entrada do modelo do que a outras, dependendo da tarefa em questão. (VASWANI et al., 2017)

O mecanismo de atenção da arquitetura Transformer usa três tensores para calcular o *score* de atenção de cada palavra em relação a todas as outras palavras da sequência de entrada. Os três tensores são denominados consulta (*query* - Q), chave (*key* - K) e valor (*value* - V).

Primeiro, a entrada é transformada em três tensores (Q, K, V) que são transformados por camadas lineares independentes. O vetor de consulta é usado para obter um score de atenção para cada palavra em relação a todas as outras palavras. O vetor chave e o vetor valor são usados para codificar informações contextuais. (VASWANI et al., 2017)

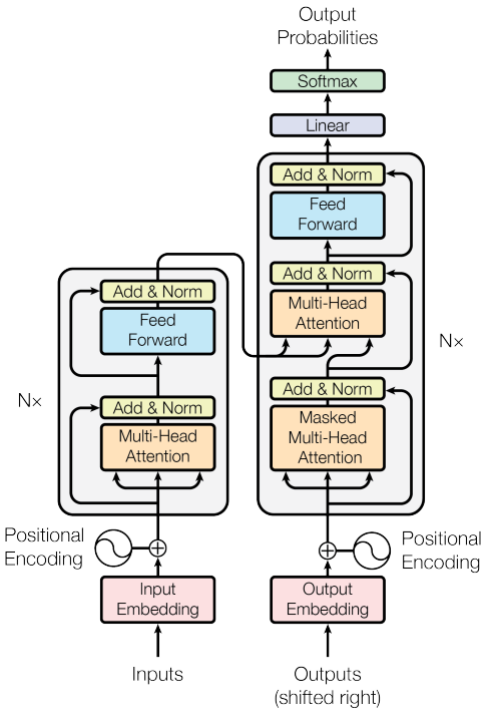


Figura 1 – Arquitetura geral de uma rede Transformer

Fonte: Vaswani et al. (2017)

Em seguida, é aplicada uma função de similaridade entre o vetor de consulta e o vetor de chave, gerando um score de atenção para cada palavra em relação a todas as outras palavras da sequência. Esses scores de atenção são normalizados por uma função *Softmax* para obter uma distribuição de pesos de atenção para cada palavra. (VASWANI et al., 2017)

Finalmente, os pesos de atenção são usados para ponderar os tensores de valor de cada palavra, gerando uma representação contextualizada para cada palavra da sequência. Essa representação contextualizada é passada para a próxima camada de codificação do modelo. (VASWANI et al., 2017)

Esse processo (descrito na figura 2) é repetido várias vezes em camadas de codificação subsequentes para permitir que o modelo capture informações de diferentes granularidades. O mecanismo de atenção é uma parte fundamental da arquitetura Transformer e é primordial para o desempenho do modelo em várias tarefas de processamento de linguagem natural.

2.2.1.2 Principais adaptações

Devido a sua popularidade a alto poder de processamento paralelo, a arquitetura Transformer vem sendo aplicado em diversos modelos, otimizando ainda mais a rede original, incluindo mais camadas e concretizando o pré treinamento para modelos com

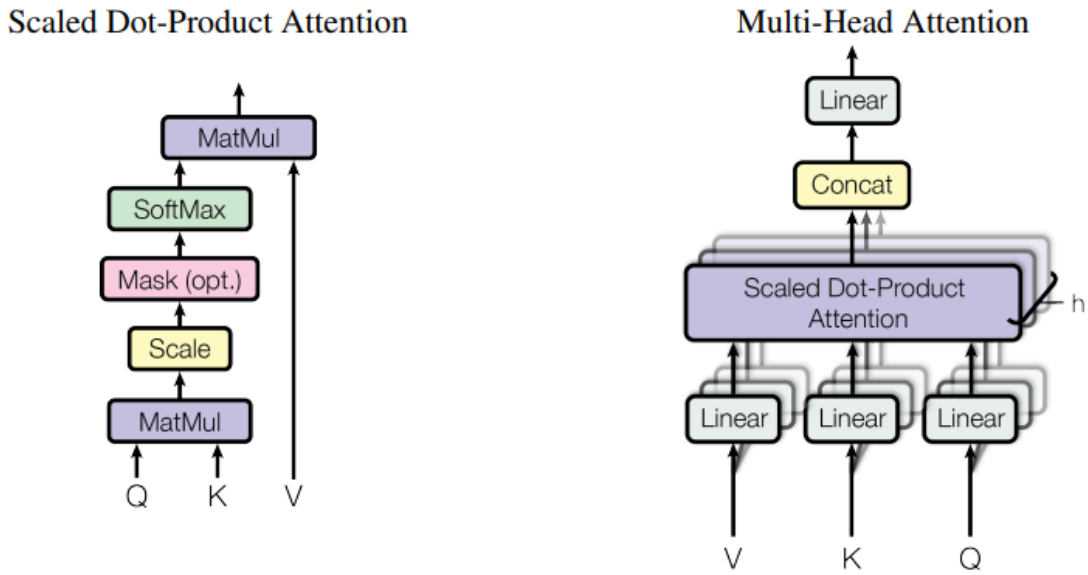


Figura 2 – (esquerda) Cálculo para matriz de atenção dos tensores *Q, K, V*. (direita) Múltiplas camadas de atenção que executam em paralelo.

Fonte: Vaswani et al. (2017)

bilhões de parâmetros.

2.2.1.2.1 GPT-3

Uma importante adaptação da arquitetura original do transforme é o modelo pré treinado GPT-3.

Enquanto a rede Transformer original é composta por algumas camadas de codificador e decodificador, o GPT-3 tem 175 bilhões de parâmetros, o que torna a rede muito mais complexa e poderosa em relação a sua capacidade de processamento de linguagem. Além disso, o GPT-3 utiliza um modelo de linguagem auto-regressivo, o que significa que ele gera sequências de palavras uma de cada vez, levando em consideração a distribuição de probabilidade condicional anterior de palavras já geradas. Isso permite que o GPT-3 possa gerar textos altamente coerentes e com uma aparência mais natural do que outros modelos de linguagem. (BROWN et al., 2020)

Outra diferença importante é que o GPT-3 é pré-treinado em uma grande quantidade de dados de texto, o que permite a geração de textos em uma ampla variedade de tópicos e estilos de escrita.

Em suma, a principal diferença entre a rede Transformer original e o GPT-3 é o tamanho, complexidade e a quantidade de dados de treinamento que são usados para refinar o modelo.

2.2.1.2.2 BERT

Outro modelo que se destaca como adaptação de sucesso da arquitetura original Transformer é o BERT (*Bidirectional Encoder Representations from Transformers*)

A principal diferença é que o BERT é um modelo de linguagem bidirecional, o que significa que ele pode ver todo o contexto de uma sequência de palavras para entender o significado de cada palavra em uma frase ou texto. Isso é diferente da arquitetura original, que é unidirecional e só pode considerar o contexto anterior de uma palavra em uma sequência. (DEVLIN et al., 2018)

Outra diferença importante é que o BERT é pré-treinado em duas tarefas diferentes: a tarefa de máscara de preenchimento (*masking*) e a tarefa de predição da próxima sentença. A tarefa de máscara de preenchimento envolve a substituição de algumas palavras em uma frase por um token especial *[MASK]* e o modelo é treinado para prever a palavra original a partir do contexto. A tarefa que prever a próxima sentença envolve o treinamento do modelo para prever se uma segunda sentença é uma continuação lógica da primeira. (DEVLIN et al., 2018)

Essas tarefas de pré-treinamento ajudam o BERT a entender melhor as relações entre as palavras dada uma sequência e auxilia na captura do significado e contexto. O BERT também é capaz de realizar várias tarefas de linguagem diferentes, incluindo classificação de texto, identificação de entidades nomeadas e resposta a perguntas, com desempenho competitivo em relação a outros modelos.

2.2.2 Aplicações na visão computacional

O artigo *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* Dosovitskiy et al. (2020) propõe a aplicação da arquitetura Transformer em tarefas de reconhecimento de imagens, mostrando que a mesma arquitetura que revolucionou o processamento de linguagem natural pode ser adaptada com sucesso para outras áreas.

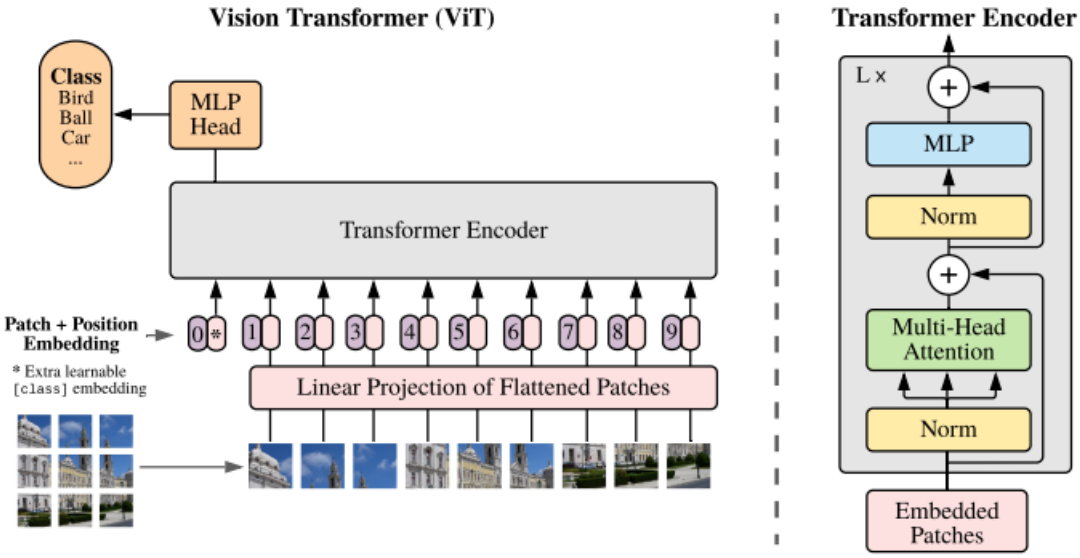
A ideia principal é tratar as imagens como uma sequência de patches (recortes retangulares da imagem original) e, em seguida, aplicar a arquitetura para gerar uma representação compacta da imagem. Isso é feito em duas etapas: a primeira etapa é a extração de características dos recortes de imagem e a segunda é a agregação das características em uma representação global da imagem. (DOSOVITSKIY et al., 2020)

Na primeira etapa, a arquitetura Transformer é usada para extrair características dos recortes de imagem em uma sequência de vetores. Cada recorte é tratado como uma palavra na sequência e a arquitetura é usada para gerar uma representação contextualizada de cada recorte, considerando os demais. Isso significa que a arquitetura dotada de seu mecanismo de atenção é capaz de capturar informações contextuais de diferentes

partes e, assim, gerar uma representação mais rica e informativa de cada pequeno grupo. (DOSOVITSKIY et al., 2020)

Na segunda etapa, as características geradas a partir dos recortes são agregadas em uma única representação global da imagem.O resultado desse agrupamento de recortes é uma camada linear, onde cada subgrupo representa uma matriz de 1x1. Essa representação global é então usada como entrada para uma camada classificadora, que atribui um rótulo à imagem, como representado pela figura 3 (DOSOVITSKIY et al., 2020).

Figura 3 – Apresentação do modelo ViT.



A imagem é dividida em conjuntos de tamanho fixo, acrescidas do seu posicionador e linearizadas. O resultado é aplicado às camadas de codificação e depois aplicadas a uma camada de classificação. Fonte: Dosovitskiy et al. (2020)

O artigo mostrou que essa abordagem, denominada ViT (*Vision Transformer*), supera as arquiteturas tidas como referência em várias tarefas de classificação de imagem em grande escala, como o reconhecimento de objetos em imagens da base de dados ImageNet, sugerindo que a arquitetura Transformer pode ser aplicada com sucesso em outras áreas além do processamento de linguagem natural.

A arquitetura Transformer aplicada a visão computacional possui apenas camadas codificadoras, estas suficientes para a tarefa de extração de características e classificação de uma imagem de entrada. Como o intuito da rede é apenas apresentar uma distribuição de probabilidades das N possibilidades para uma imagem, a camada de decodificação se torna desnecessária. Em problemas de visão computacional destinados a geração de novas imagens, a inserção de camadas decodificadores serviria para este propósito, como é feito no processamento de linguagem natural, onde novas representações são geradas no processo de decodificação.

2.3 Impactos do tamanho da rede no processo de treinamento

No livro *"Deep Learning Book"*, Goodfellow, Bengio e Courville (2016), é discutido como o número de parâmetros de uma rede neural profunda pode afetar os custos computacionais envolvidos no treinamento e na inferência da rede.

Os autores afirmam que o número de parâmetros é um fator importante na determinação do tempo e da memória necessários para o treinamento e a inferência de uma rede neural profunda. Quanto mais parâmetros uma rede tiver, mais tempo e memória serão necessários para calcular os gradientes e fazer as atualizações dos parâmetros durante o treinamento, bem como para realizar a inferência durante a etapa de teste.

Além disso, redes com muitos parâmetros podem ser mais suscetíveis a problemas de *overfitting*, enviesando o modelo para um determinado conjunto de dados e tornando-o incapaz de prever novos resultados, o que pode exigir mais iterações de treinamento e, portanto, mais tempo de computação.

Para lidar com o problema de redes com muitos parâmetros, o livro sugere várias técnicas de regularização, como regularização, *dropouts* e normalização de camadas. Essas técnicas podem ajudar a reduzir o número de parâmetros da rede, tornando o treinamento mais eficiente e ajudando a evitar o *overfitting*.

2.4 Computação de Alto Desempenho

Computação de alto desempenho (HPC, do inglês *High Performance Computing*) é uma área da computação que se dedica ao desenvolvimento e uso de sistemas de computação com alta capacidade de processamento para lidar com problemas complexos e intensivos em determinados recursos computacionais. Esses sistemas podem ser compostos por uma grande quantidade de processadores trabalhando em paralelo, memória de alta velocidade, redes de alta velocidade e armazenamento em massa de dados. (STERLING; ANDERSON; BRODOWICZ, 2017)

A HPC tem aplicações em diversas áreas, como previsão do tempo, modelagem climática, simulação de sistemas complexos, genômica, análise de dados em larga escala, entre outras. A computação de alto desempenho é também um campo em constante evolução, com novas tecnologias e técnicas sendo desenvolvidas para aumentar ainda mais a capacidade de processamento dos sistemas.

Um exemplo de tecnologia usada em HPC é o uso de aceleradores, como GPUs (Graphics Processing Units), que podem oferecer um aumento significativo no desempenho de cálculos intensivos. Além disso, o uso de arquiteturas de computação distribuída e a implementação de algoritmos paralelos podem melhorar ainda mais o desempenho de sistemas HPC. (STERLING; ANDERSON; BRODOWICZ, 2017)

A distribuição do processamento entre sistemas distribuídos pode ser outro componente importante para otimizar o tempo de resposta de uma dada aplicação.

A biblioteca MPI é uma interface padronizada para comunicação entre processos paralelos em sistemas distribuídos. Ela permite que os processos troquem mensagens entre si e coordena sua execução de forma eficiente. A principal vantagem do MPI é que ele pode ser usado em diferentes arquiteturas de hardware e sistemas operacionais, tornando-o uma ferramenta amplamente utilizada em HPC.

O livro *High Performance Computing: Modern Systems and Practices* Sterling, Anderson e Brodowicz (2017) apresenta vários exemplos de como usar o MPI para programação paralela em HPC. Um exemplo é a paralelização de algoritmos de processamento de imagens, onde a imagem é dividida em várias partes e cada parte é processada por um processo separado. Outro exemplo é a simulação de sistemas físicos complexos, onde a simulação é dividida em várias partes que são executadas em paralelo.

No livro, também são discutidos os desafios da programação paralela com MPI, como o *overhead* de comunicação e a necessidade de sincronização entre os processos. São apresentadas estratégias para minimizar esses desafios, como a sobreposição de comunicação com computação e a utilização de algoritmos eficientes para distribuição de dados.

Se tratando de processamento em uma máquina única, uma das melhorias que podem ser aplicadas ao problema proposto é a utilização do OpenMP, uma interface de programação de aplicativos de memória compartilhada que permite a paralelização de código em sistemas multiprocessados e *multicore*. Ela é uma ferramenta amplamente utilizada em HPC, pois oferece uma maneira relativamente simples de explorar a paralelismo em código existente, sem exigir uma grande mudança na arquitetura do programa.

Outro exemplo proposto no livro de Sterling, Anderson e Brodowicz (2017) é a paralelização de algoritmos de processamento de imagens, onde o processamento de cada pixel é executado em paralelo em diferentes núcleos. Outro exemplo é a paralelização de algoritmos de simulação, onde várias partes do algoritmo são executadas simultaneamente em diferentes núcleos.

A aplicação de uma computação de alta performance está relacionado a uma gama de características aplicadas ao problema alvo, desde a linguagem de programação utilizada, especificações do *hardaware*, bibliotecas que podem ser incorporadas, balanceamento de carga, gerenciamento de memória, comunicação entre processo, etc. Analisar todas estas variáveis exige um time altamente capacitado, tempo e recursos financeiros, coisas que nem sempre estão disponíveis.

2.4.1 Linguagem de domínio especifico - Devito

Devito é um framework de código aberto de alto desempenho para soluções numéricas baseadas em equações diferenciais parciais (PDEs) e equações de onda. Ele permite que os usuários escrevam equações matemáticas em uma forma simples e concisa, e o framework gera automaticamente o código otimizado para a CPU ou GPU. Devito permite acelerar a solução de problemas de PDEs e equações de onda em várias ordens de grandeza em comparação com abordagens tradicionais baseadas em códigos escritos e otimizados manualmente. (LONDON, 2016)

O Devito surgiu como uma iniciativa de pesquisa focada em otimização do processamento de dados sísmicos para renderização de imagens. A proposta era oferecer aos pesquisadores um conjunto de operações finitas que podem ser escritas em stencil (auxiliado pela biblioteca python SymPy) e a partir dessas equações, gerar no seu compilador em tempo real, códigos otimizados na linguagem C/C++, oferecendo configurações arquiteturais para processamento em múltiplas *"threads"* e processamento distribuído com o MPI. (LONDON, 2016)

O núcleo orquestrador das gerações e compilações de código do framework é expresso pela classe *"Operator"* que carrega as três funções principais para o resultado final. A operação de geração de código de baixo nível, a compilação em tempo real (*"Just in time"* - JIT) e a própria execução da operação. (LUPORINI et al., 2020)

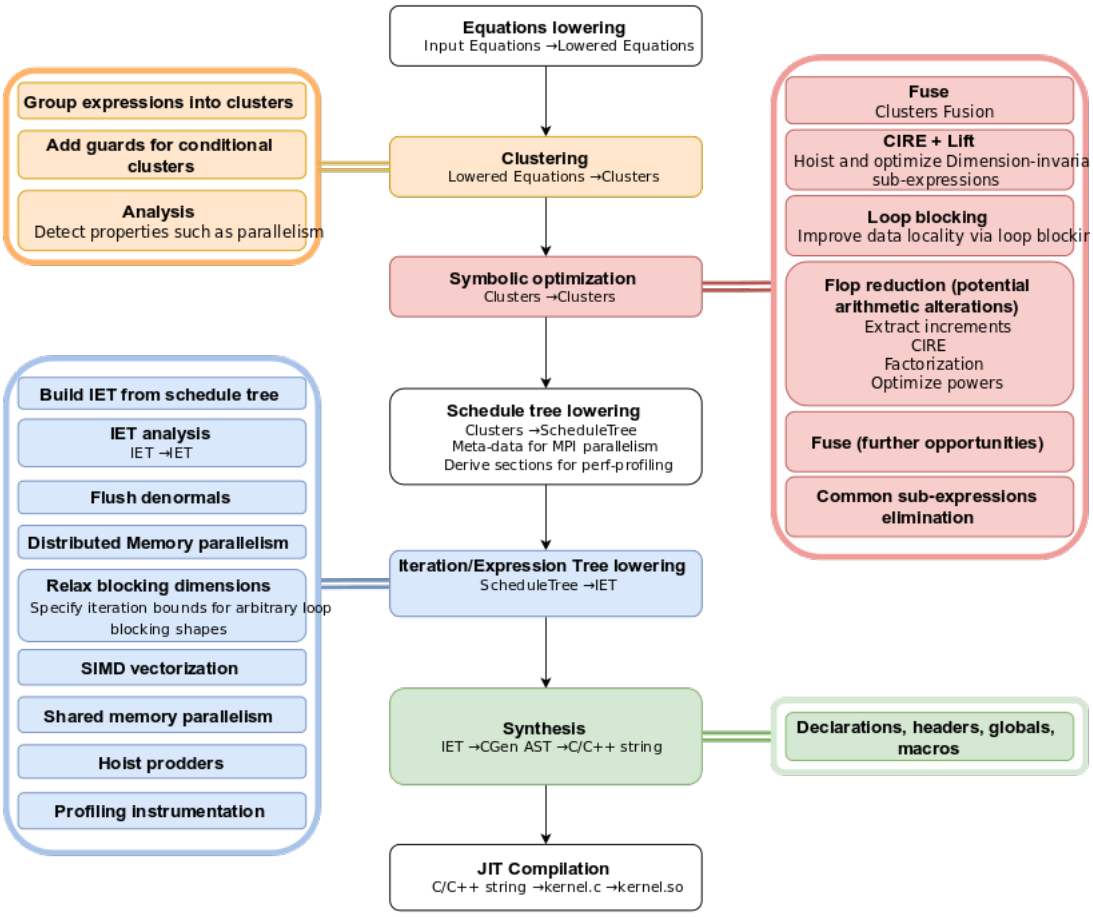
A figura 4 ilustra de cima para baixo os passos sumarizados da classe Operator do Devito.

O operador Devito cria código de baixo nível, utiliza o compilador JIT e executa o código. O primeiro passo é desmembrar a equação, convertendo as funções Devito em matrizes que armazenam um ponteiro para essa função original. Em seguida, é realizada uma análise local nas equações em níveis mais baixos para coletar informações úteis para a execução do operador, como as dimensões, as funções de entrada e saída e os espaços de iteração e de dados. (LUPORINI et al., 2020)

O agrupamento reúne equações com o mesmo espaço de iteração. A primeira otimização automática de desempenho é a otimização simbólica, que reduz a contagem de operações de agrupamentos, influenciando, indiretamente, a estrutura final de laços. Isso é feito através do *Devito Symbolic Engine*, que elimina sub-expressões comuns, fatoriza e extrai sub-expressões que satisfazem certas condições. As expressões são abaixadas ainda mais para uma árvore de iteração/expressão (IET). (LUPORINI et al., 2020)

Essas expressões são analisadas e otimizadas ainda mais pelo Devito Loop Engine (DLE). Os principais passos para essas otimizações são o bloqueio de laço, a vetorização de instrução única, múltiplos dados (SIMD) e o paralelismo em memória compartilhada. Finalmente, o código C é gerado inspecionando a IET e traduzindo-a para uma *string*. O

Figura 4 – Arquitetura da classe Operator



Fonte: Luporini et al. (2020)

código C é compilado em tempo de execução e carregado no ambiente python. (LUPORINI et al., 2020)

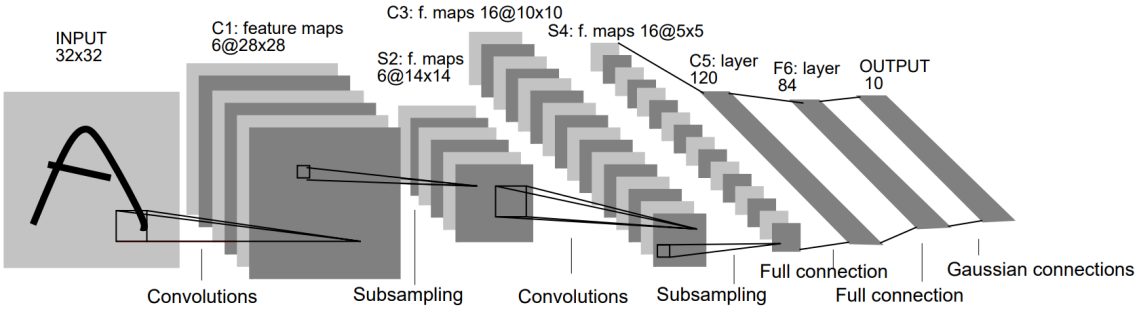
As otimizações realizadas pelo Devito podem variar de acordo com a arquitetura da máquina, recursos disponíveis e parâmetros de configuração dos otimizadores. As otimizações podem ser a nível de *"loops"*, onde o Devito utiliza técnicas avançadas como a vetorização e a paralelização para acelerar a execução dos códigos. Otimizações de acesso de memória também podem ser aplicados, onde o framework organiza a memória em blocos, para garantir eficiência no acesso aos dados a serem manipuladores pelas equações diferenciais, reduzindo o número de escrita e leitura na memória da máquina.

2.4.2 Framework Joey

Elaborado a partir de um conjunto de classes que implementam as equações através do Devito, o framework Joey surgiu como uma proposta de otimização para a fase de treinamento e validação de redes neurais. Focado na implementação de um conjunto de

cinco camadas para problemas de aprendizado de máquinas, a rede construída, em seus experimentos originais, mostrou resultados melhores para treinamento de imagens grandes (maiores que 256px) em comparação com a mesma rede implementada no framework PyTorch. (CHATZITHEOKLITOS, 2020)

Figura 5 – Visão geral da rede LeNet5



Fonte: LeCun et al. (1989)

A rede construída em ambos os frameworks foi a *LeNet5*, uma rede convolucional (conforme descrita na figura 5) e a performance resultante foi avaliada baseada na permutação do tamanho da imagem de entrada, tamanho do conjunto de entrada (quantidade de imagens de um lote) e o número de iterações. Os resultados podem ser observados nas figuras 6 e 7.

Para a construção da rede *LeNet5*, as seguintes camadas foram implementadas no framework Joey:

- Camada de Convolução bidimensional
- Camada de Pooling bidimensional (Agrupamento)
- Camada Linear bidimensional (*Fully Connected*), disponível também com a aplicação da função *Softmax*
- Camada Flat (Achatamento)

A estrutura de implementação de todas as classes seguem o mesmo princípio, herdam da classe base *Layer* e durante a construção da rede geral (agrupamento de N camadas), cada classe retorna uma tupla de *Function*'s oriunda do devito.

As camadas podem funcionar de forma isolada ou através da sua alocação na classe Net, disponível no Joey. A classe é responsável por atribuir a cada camada seus pesos e bias iniciais, definir as equações em uma ordem sequencial de todas as camadas,

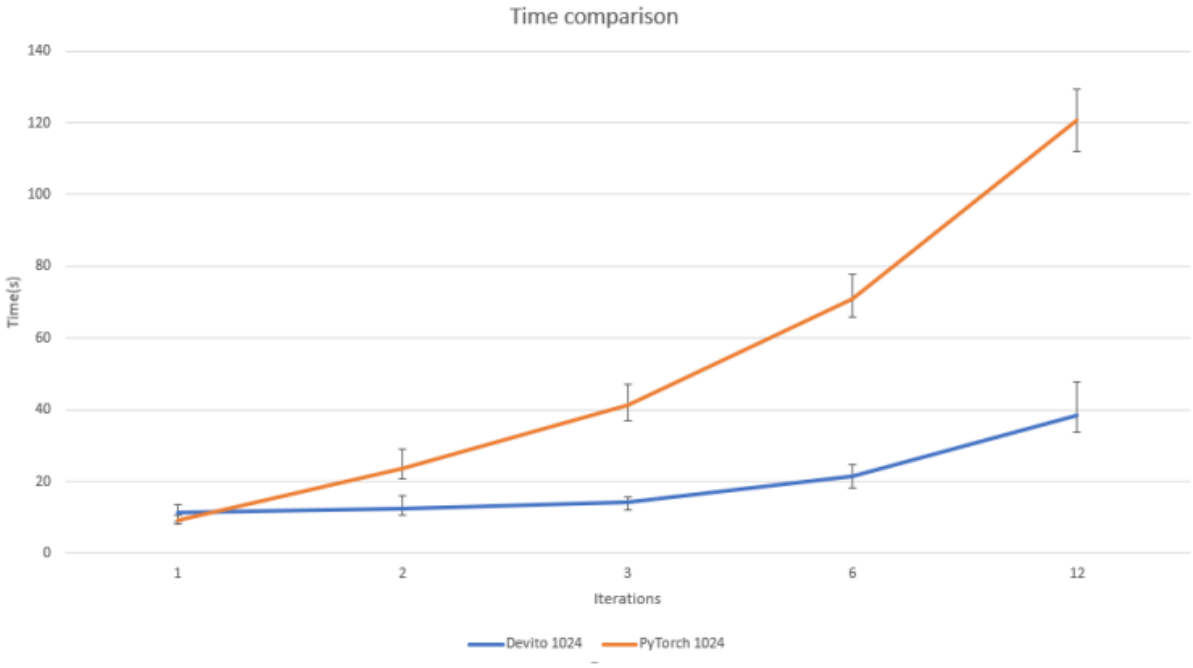
gerando na própria rede uma equação única que será aplicada na classe *Operator* do Devito, produzindo um código otimizado de acordo com os parâmetros definidos.

Apesar de resultados promissores, mesmo para camadas mais simples e uma rede não complexa, o framework Joey não está disponível para ser instalado via o gerenciador de pacotes do python, utilizamdo o comando *pip install joey*.

O framework teve sua publicação em um repositório da devitocodes (proprietária do código Devito) em 2019, sendo esta sua versão disponível atualmente, sem atualizações posteriores.

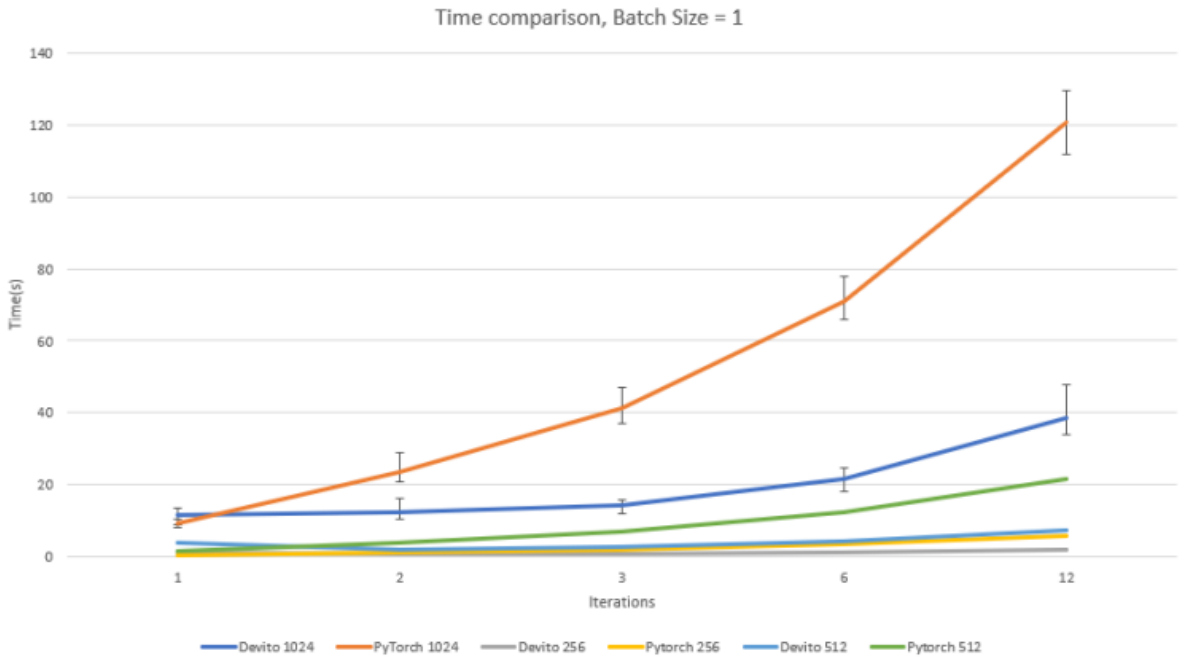
Analisando os resultados oferecidos pelo framework Joey para a rede *LeNet* em comparação com o framework Torch, especialmente para uma entrada de dados maior, é esperável que outras redes que sigam a mesma lógica de entrada de dados de forma sequencial, projetem resultados parecidos.

Figura 6 – Comparativo de tempo, variação no número de iterações para imagens de 1024px



Fonte: Chatzitheoklitos (2020)

Figura 7 – Comparativo de tempo, variação no tamanho de imagem e número de iterações



Fonte: Chatzitheoklitos (2020)

3 Desenvolvimento

3.0.1 Experimentos iniciais

Para introdução aos métodos do Joey e seu próprio funcionamento, os experimentos iniciais utilizando a rede LeNet foram reproduzidos, seguindo a própria implementação modelo que está disponível no repositório oficial do framework[1].

Os experimentos foram realizados através do Google Colab, ambiente capaz de executar scripts em python, e instalar pacotes de dependências em momento de execução. As especificações da máquina disponível são:

- Processador dual core com frequência de 2.2GHz de clock[2]
- 13 GB RAM
- 107 GB de armazenamento HDD

Para inicialização das camadas disponíveis do framework Joey e construir rede *LeNet*, a dependência principal para sua execução, Devito, foi instalada através do gerenciador de pacotes do python. A versão utilizada do framework foi a 4.2.3, sendo esta a última versão totalmente compatível com o Joey.

Para a construção da rede, as camadas foram inicializadas de forma isolada, contendo as especificações de tamanho de entrada, pesos e tamanho do filtro no caso das camadas de convolução. Cada camada descrita no Joey contém um conjunto de equações que definem as operações a serem realizadas pelo *Operator* do Devito, tanto para a fase de treinamento, quanto para a fase de validação.

Após a instanciação de cada camada isolada, a classe *Net* realiza o agrupamento dessas funções de forma ordenada, a fim de se obter a saída desejada. Para a fase de validação, cada camada é iniciada com o valor $= 0$ em seu tensor de resultado, garantindo que a cada iteração pela rede, o resultado anterior não seja somado ao resultado atual. Cada camada retorna para a rede geral suas equações que produzirão o resultado a partir de uma entrada, este resultado é atribuído por meio das equações à entrada da camada seguinte. Esta fase é nomeada *forward*.

Para a fase de treinamento e regulação dos pesos e viés, a rede atribui para a última camada, o erro produzido pela saída rede, que de forma inversa a fase de validação, atribui

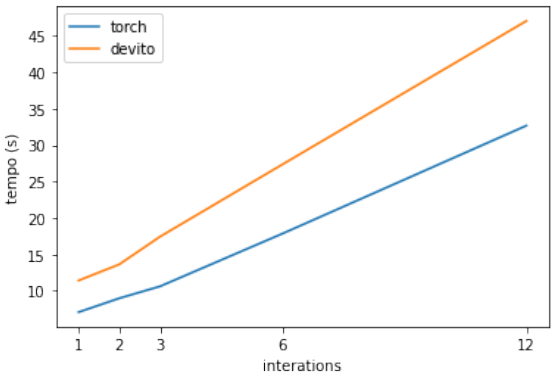
[1] Disponível em: <https://github.com/devitocodes/joey>.
[2] O processador alocado é definido no momento de início do programa, podendo ser um Intel Xeon ou um AMD EPYC.

à camada anterior o erro produzido pela camada mais a frente até a primeira camada. Esta fase é conhecida como *backpropagation*.

Para treinamento de ambas as redes, a base de dados MNIST LeCun et al. (1989) foi utilizada. O pacote conta com 60.000 amostras de imagens referentes aos dígitos do 0 ao 9. Como ambas as redes são iniciadas com pesos aleatórios que variam de -0.5 a 0.5, os pesos foram igualados em cada modelo antes de cada treinamento para garantir que as duas partissem de um mesmo ponto.

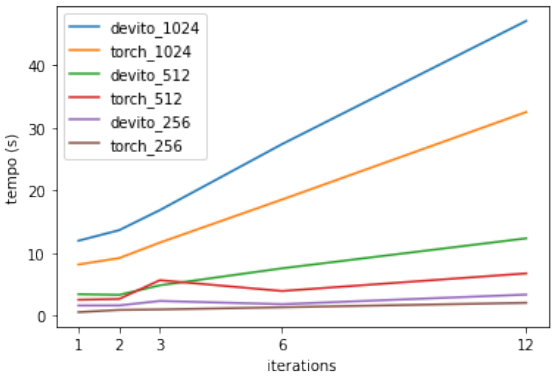
Os resultados obtidos para os experimentos referência do relatório do Joey, explicitados nas imagens 6 e 7, foram diferentes dos esperados, mesmo analisando de forma proporcional, a eficiência de tempo de treinamento obtida no Joey, utilizando as operações do Devito foi inferior aos resultados do framework PyTorch, como seguem nas Figuras 8 e 9.

Figura 8 – LeNet Testes locais - Comparação de tempo, variação no número de iterações - Imagem 1024px



Fonte: o autor

Figura 9 – LeNet Testes locais - Comparação de tempo, variação no tamanho de imagem e no número de iterações



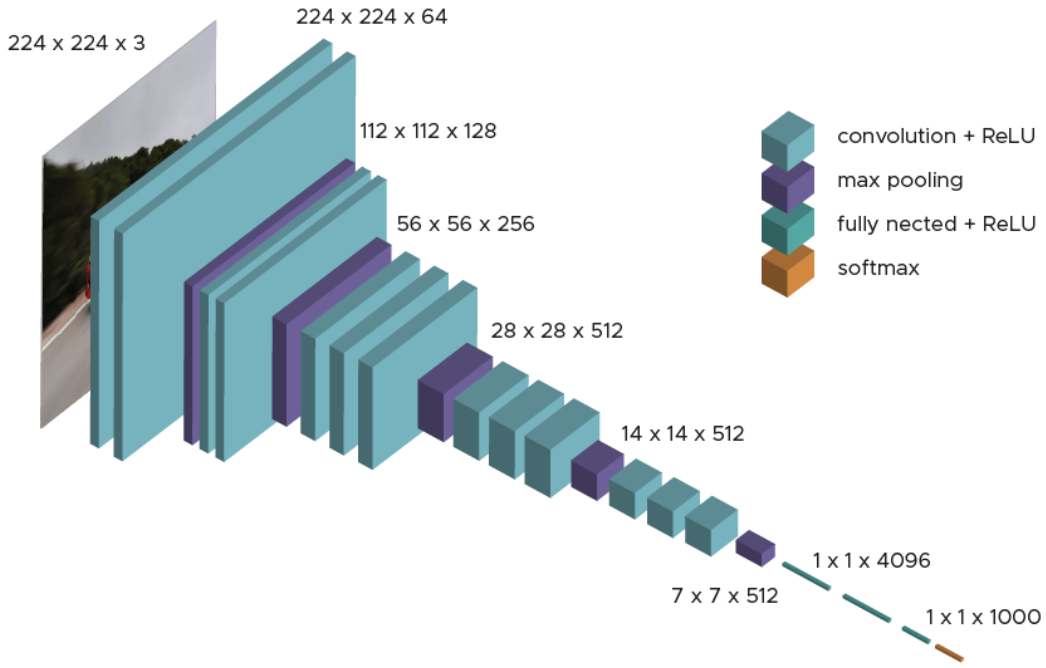
Fonte: o autor

Como os resultados foram incompatíveis com os apresentados pelo Chatzitheoklitos (2020), uma outra implementação de CNN que possui camadas similares a *LeNet* foi submetida a testes. A rede em questão é a *Very Deep Convolutional Network* (VGG), também destinadas a problema de classificação de imagem, porém com mais camadas escondidas na sua implementação.

A VGG é uma rede neural convolucional profunda, proposta no artigo *"Very Deep Convolutional Networks for Large-Scale Image Recognition"* Simonyan e Zisserman (2015). Ela é composta por camadas convolucionais com filtros pequenos de 3x3, seguidas por camadas de pooling, e finalmente por camadas totalmente conectadas. A arquitetura da VGG é conhecida por ter um número relativamente grande de camadas (até 19 camadas) e uma profundidade maior do que outras redes convolucionais na época em que foi proposta.

A figura 10 [3] ilustra um exemplo de implementação da rede VGG16.

Figura 10 – Visão geral da rede VGG16



Fonte: site datascientest.com.

A construção desta rede utilizando o Joey no seu atual estado foi possível por não existirem camadas diferentes das atuais, sem necessidade de novas implementações ou ajustes. Os testes para treinamento da rede foram aplicados seguindo a mesma base MNIST e mesma variação de iterações dos testes realizados com a rede *LeNet*.

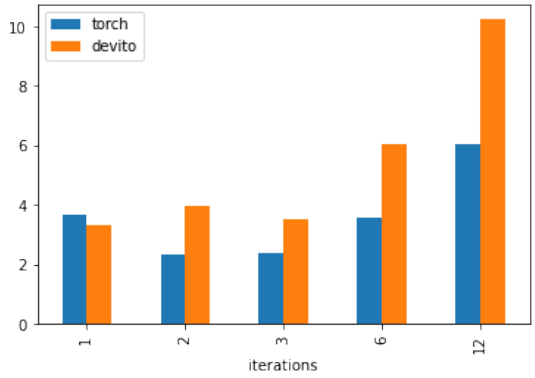
Os resultados de tempo de processamento obtidos na fase de treinamento para esta rede também foram maiores que a mesma rede elaborada a partir do framework PyTorch, como seguem nas figuras 11 e 12. O único teste que reportou um tempo de processamento menor no Joey foi a iteração unitária para imagens de 1024px.

3.0.2 Equações utilizando o framework Devito

Para o desenvolvimento de novas camadas para o framework Joey, as equações simbólicas do Devito foram utilizadas. Os valores dos dos tensores que serão manipulados pelas operações matriciais, são armazenados em um objeto da classe *Grid*, que é criado com as especificações de suas dimensões. O objeto *Grid* é então atribuído a um objeto da classe *Function* e através dessa implementação, o Devito é capaz de gerar os códigos otimizados das equações.

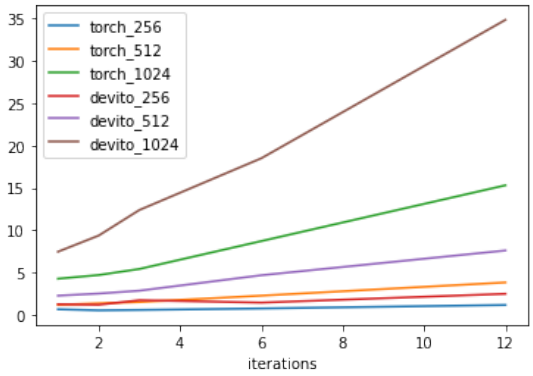
[3] Disponível em <https://datascientest.com/es/vgg-que-es-este-modelo-daniel-te-lo-cuenta-todo>

Figura 11 – VGG Testes locais - Comparação de tempo, variação no número de iterações - Imagem 1024px



Fonte: o autor

Figura 12 – VGG Testes locais - Comparação de tempo, variação no tamanho de imagem e no número de iterações



Fonte: o autor

A classe *Operator* recebe como parâmetro uma lista de equações que deverão ser executadas em ordem de listagem. A cada dimensão distinta identificada entre as *Funtions*, um nível de iteração é gerado no código em C. Para fins de esclarecimento, dado o exemplo de uma $FunctionA(dimensões : (a, b))$ e $FunctionB(dimensões : (a, c))$, o código resultante da classe *Operator* terá 3 níveis de iteração. Durante o processo de otimização realizado pela DLE, o número de iterações pode ser reduzido.

As equações definidas no Devito podem ser do tipo $Eq()$ ou $Inc()$, a diferença entre as duas é a operação realizada no ultimo nível de iteração. Para as $Eq()$ o compilador gera um código de atribuição, paras as $Inc()$, um código de incrementação é gerado. Entender o comportamento destes dois componentes é fundamental para a produção de novas camadas.

3.0.3 Implementação da ViT e adaptações do Joey

Tomando como referência a implementação[4] da arquitetura ViT do artigo Dosovitskiy et al. (2020), a rede foi reproduzida em ambiente local para entendimento de todo seu funcionamento e identificação das camadas e operações faltantes no framework Joey.

De forma ampla, a rede possui três componentes principais, a camada *MultiHeadAttention*, a *VisionEnconder* e a classe de agregação principal *ViT*.

Para a construção de uma Transformer que utiliza das operações do Devito, as seguintes camadas foram construídas, em abiente local, utilizando a IDE PyCharm.:

- *FullyConnected3d*

[4] Disponível em <https://github.com/UdbhavPrasad072300/Transformer-Implementations/blob/main/transformer_package/models/transformer.py>

- *LayerNorm2d*
- *LayerNorm3d*

A função de normalização de dados *Softmax* foi isolada em um conjunto de equações, que anteriormente estava conetada ao final do resultado produzido pela camada *FullyConnected* do Joey.

Um conjunto de equações que definem as operações realizadas pela função *einsum* Team (2021), também foi incorporado ao framework, possibilitando que operações neste formato sejam aplicadas a tensores de 4 dimensões.

Duas outras funções para manipulação de dimensões dos tensores foram aplicadas ao framework, bem como a construção de uma função *Dropout*. A construção destas novas camadas e operações são detalhadas a seguir.

Seguindo de forma sequencial a entrada de dados pela rede ViT, a primeira camada faltante identificada no framework Joey, é uma adaptação da camada existente *FullyConnected* que na implementação original atendia somente entradas de dimensão tamanho 2. A adaptação incluiu uma nova dimensão às equações originais, resultando em mais um nível iteração sobre os tensores de entrada.

Para fins de explicação, dada uma entrada de dimensões (P, N, M), e desejada saída de tamanho (P, N, K), a camada gera pesos randômicos de dimensão (K, N), respeitando assim as regras de multiplicação matricial, e bias de dimensão (K). Traduzindo para as equações do Devito, a operação pode ser expressa pela equação 3.1 A geração da saída é definida pelo algorítimo 1.

$$\begin{aligned} Equações = [& \\ & Inc(Saída[P, N, K], Entrada[P, N, M] * Pesos[K, N]), \\ & Inc(Saída[P, N, K], Bias[K]) \\ &] \end{aligned} \tag{3.1}$$

Para a criação de uma rede neural, é essencial compreender como os tensores são manipulados durante o processamento. Neste contexto, outra camada não implementada na versão original do framework Joey é a *LayerNorm* Ba, Kiros e Hinton (2016). O intuito desta cama em suma é aprimorar o tempo de processamento durante a fase de testes e de treinamento de uma rede neural aplicando a normalização por média e variância da saída de uma camada anterior, reduzindo desta forma a entrada da camada posterior e minimizando os problemas de *overfiting*.

A saída produzida pela camada de normalização possui mesmo tamanho e formato do tensor de entrada denominado por x. O tensor de pesos (γ) e bias (β) possuem tamanho

Algorítimo 1 FullyConnectd3d

```
    função FullyConnected3d(Entrada)
2:                                          ▷ Entrada nas dimensões (P, N, M)
      Saída ← 0                             ▷ Saída nas dimensões (P, N, K)
4:    para p ← P_1 até P_P faça
        para n ← N_1 até N_N faça
6:        para k ← K_1 até K_K faça
            Saída[p][n][k] ← 0
8:          para m ← M_1 até M_M faça
              Saída[p][n][k] ← Saída[p][n][k] + Pesos[k][m] * Entrada[p][n][m]
10:         fim para
            Saída[p][n][k] ← Bias[k] + Saída[p][n][k]
12:       fim para
        fim para
14:   fim para
      retorna Saída
16: fim função
```

igual a ultima dimensão de x. A contante ϵ possui valor padrão de 1^{-6}, servindo como constante de estabilização numérica. A saída y que é produzida pela camada é descrita pela equação 3.2, onde a média sobre as últimas N dimensões de x ($E[x]$) é subtraída de x e seu resultado é dividido pela raiz quadrada da variação padrão de $x + \epsilon$ ($\sqrt{Var[x] + \epsilon}$). O resultado é multiplicado pelo tensor unidimensional γ e acrescido do tensor unidimensional β.

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta \tag{3.2}$$

As operações feitas via Devito ocorrem sobre tensores com tamanho pré definidos e imutáveis em tempo de execução, sendo assim, para a elaboração das equações que viabilizam a construção da camada de normalização, tensores adicionais que armazenam o resultado de forma temporária da somatória e da variação padrão foram criados, as equações do Devito podem ser expressas como seguem na equação 3.3

$$
\begin{aligned}
Equações = [\\
&Inc(Somatório[B, M, 1], Entrada[B, M, N]), \\
&Eq(Média[B, M, 1], Somatório[B, M, 1]/N), \\
&Inc(VariçãoPadrão[B, M, 1], (Entrada[B, M, N] - Média[B, M, 1])^2) \\
&Eq(VariçãoPadrão[B, M, 1], \sqrt{VariçãoPadrão[B, M, 1]/N}) \\
&Eq(Resultado[B, M, N], Pesos[N] * (Entrada[B, M, N] - Média[B, M, 1])) \\
&Eq(Resultado[B, M, N], Resultadp[B, M, N]/(VariçãoPadrão[B, M, 1] + \epsilon) \\
&Inc(Resultado[B, M, N], Bias[N]) \\
&]
\end{aligned}
\tag{3.3}
$$

Outra camada presente na implementação é a *Dropout* Goodfellow, Bengio e Courville (2016), a camada em questão recebe como parâmetro o percentual de corte (dp) das saídas produzidas pela camada anterior. A camada inicia um tensor de pesos de tamanho igual ao tensor de entrada, distribui de forma randômica valores $= 0$, que por sua vez é multiplicado pela entrada, produzindo a saída com valores zerados aleatoriamente. a técnica é mais uma ferramenta para evitar o *overfiting*.

Durante a manipulação de dados dos tensores, a camada *MultiHeadAttention* da implementação do ViT, realiza uma transformação de expansão dos tamanhos de uma determinada entrada, produzindo um tensor de mesmo tamanho, mas com diferentes dimensões ($expanção = d + 1$).

A manipulação deste tensor de entrada visa separar em J cabeças de tamanho L a ultima dimensão do tensor de entrada, neste sentindo, o valor denotado para $J * L$ deve ser igual a M (ultima dimensão da entrada). A expansão é definida pelo algorítimo 2

A camada de atenção da arquitetura Transformer é uma parte fundamental para a rede, produzindo os scores de atenção para cada palavra em relação a sentença geral, para a ViT, o conceito se mantém o mesmo, onde palavras são representadas por recortes de imagens e a sentença se refere a imagem total.

A função *einsum* Team (2021) é uma operação matemática que permite a manipulação de tensores em várias dimensões. Ela foi introduzida na biblioteca NumPy, em 2011, e desde então se tornou uma ferramenta fundamental para muitas aplicações científicas e de engenharia. A aplicação dessa contração de tensores é parte da camada de *MultiHeadAttention*, onde os valores dos tensores $Q(query), K(key)$ e $V(value)$ produzem o valor de atenção resultante da camada.

O algorítimo recebe como entrada uma expressão algébrica que define a operação a ser realizada, juntamente com os tensores que serão manipulados. A expressão algébrica

Algorítimo 2 Enxpand3dto4d

```
    função EXPAND3DTO4D(Entrada)
2:                                          ▷ Entrada nas dimensões (P, N, M)
      Saída ← 0                             ▷ Saída nas dimensões (P, N, J, L)
4:    para x ← P_1 até P_P faça
        para y ← N_1 até N_N faça
6:        para z ← J_1 até J_J faça
            para w ← L_1 até L_L faça
8:            Saída[x][y][z][w] ← Entarda[x][y][(z * J) + w]
            fim para
10:       fim para
        fim para
12:   fim para
      retorna Saída
14: fim função
```

consiste em uma sequência de caracteres que indicam as dimensões dos tensores e as operações matemáticas a serem realizadas. A partir dessa expressão, a função calcula o resultado da operação e retorna um novo tensor.

A função *einsum* é particularmente útil em problemas que envolvem a contração de tensores, ou seja, a redução do número de dimensões de um tensor por meio da multiplicação de seus elementos. Ela também pode ser usada para realizar operações de soma, multiplicação e outras operações matemáticas em tensores em várias dimensões.

Para a elaboração da função de soma utilizando as operações do Devito, é importante identificar as dimensões do resultado esperado, comparando com as dimensões dos tensores de entrada. A primeira operação *einsum* aplicada na camada *MultiHeadAttention*, opera sobre os tensores Q e K, ambos redimensionados. Dado a entrada $Q(b, q, h, e)$ e $K(b, k, h, e)$, para produzir a saída $S(b, h, q, k)$, é necessário um iteração em todas dimensões de S acrescido de uma iteração sobre a dimensão a ser reduzida em ambos tensores de entrada $Q[e], K[e]$.

Durante o processo de geração do código otimizado na classe *Operator* do Devito, a etapa de verificação de dependência cíclica procura por dimensões que possam interferir no cálculo das operações matriciais. A saída S possui sua segunda dimensão de tamanho h, e em ambos os tensores de entrada Q e K, a posição denotada por h é inerente a terceira dimensão. Dado essa incompatibilidade do Devito em realizar operações com possível dependência cíclica. a camada h pode ser iterada de forma manual, não interferindo nos resultados esperados, como descrito na equação 3.4.

$$\forall i \in \{1, 2, 3, \ldots h\}, Equações[\,] \leftarrow Inc(S(b, i, q, k), Q(b, q, i, e) * K(b, k, i, e) \tag{3.4}$$

O código resultante para a função requerida, é expresso pelo algorítimo 3.

Algorítimo 3 Einsum

função EINSUM($Query$, Key)
▷ Query nas dimensões (B, Q, H, E)
▷ Key nas dimensões (B, K, H, E)
$Saída \leftarrow 0$ ▷ Saída nas dimensões (B, H, Q, K)
para $b \leftarrow B_1$ até B_B **faça**
para $q \leftarrow Q_1$ até Q_Q **faça**
para $k \leftarrow K_1$ até K_K **faça**
para $e \leftarrow E_1$ até E_E **faça**
$Saída[b][1][q][k] \leftarrow Saída[b][1][q][k] + Query[b][q][1][e] * Key[b][k][1][e]$
$Saída[b][2][q][k] \leftarrow Saída[b][2][q][k] + Query[b][q][2][e] * Key[b][k][2][e]$
$\vdots$
$Saída[b][h][q][k] \leftarrow Saída[b][h][q][k] + Query[b][q][h][e] * Key[b][k][h][e]$
fim para
fim para
fim para
fim para
retorna $Saída$
fim função

Após a aplicação da primeira compressão de tensores, a saída é dividida pela raiz quadrada do número de colunas do tensor *Key*. A divisão recebe uma camada *Softmax*, normalizando no último eixo os valores de forma probabilística, onde seu somatório $= 1$. A saída normalizada segue para o próxima função *einsum*, onde os valores de $S(b, h, q, l)$ são multiplicados pelos valores do tensor $K(b, l, h, d)$, produzindo a saída $Score(b, q, h, d)$. A lógica de implementação segue a mesma para a primeira função aplicada, onde para as dimensões cíclicas identificadas, a interação sobre o eixo pode ser feita de forma manual. O processo geral da entrada até a saída da camada é ilustrado pela figura 2.

A implementação original do framework Joey possibilita a utilização de camadas lineares com aplicação da função *Softmax* ao final das operações de pesos(γ) e bias(β). Para a completa implementação da rede ViT em conjunto com o Devito, as operações realizadas para normalizar a saída foram isoladas em um conjunto de operações específicas para a produção do resultado em tensores de dimensão 3 e 4.

A operação de redução de dimensões é realizada na classe *ViT*, após a entrada passar pelas operações de todas as camadas *VisionEnconder* definidas no modelo. A redução ocorre na segunda dimensão do tensor de resultante. Para um tensor *Entrada* de dimensões (A, B, C), a redução da dimensão B é definida pelo algorítimo 4

Após a implementação de todas camadas e operações descritas acima, o passo seguinte foi montar a rede de forma total. Como a implementação da classe *Net* do Joey não previa implementações de módulos complexos, somente a inicialização de camadas

Algorítimo 4 ReducaoSegundaDimensao

```
    função REDUCAO2NDDIM(Entrada)
2:                                          ▷ Entrada nas dimensões (A, B, C)
      Saída ← 0                             ▷ Saída nas dimensões (A, C)
4:    para a ← A_1 até A_A faça
          para c ← C_1 até C_C faça
6:            Saída[a][c] ← Entrada[a][b][c]
          fim para
8:    fim para
      retorna Saída
10: fim função
```

já implementadas, ajustes foram aplicados na classe para que a camada de *LayerNorm* tivesse seus pesos e bias inicializados seguindo uma lógica diferente das demais.

As classes *MultiHeadAttention* e *VisionEncoder* possuem o funcionamento semelhante ao de uma *Layer* do Joey, porém realizam operações mais complexas e envolvem mais de uma camada na sua implementação. Para garantir que estes módulos pudessem ser instanciados em um único objeto da classe *Net*, a classe base *Module* foi criada e estes dois novos módulos herdam desta classe.

A nova estrutura de arquivos e diretórios proposta para o framework Joey está representada na figura 13, com os arquivos destacados com a marcação **(novo)** indicando arquivos criados e incorporados ao framework, e os arquivos com a marcação **(modificado)** indicando que alterações foram aplicadas ao arquivo original. O arquivo *functional.py* contém as novas operações de *Dropout*, disponível em 1, 2, 3 e 4 dimensões, e a função *Softmax* para tensores de 3 e 4 dimensões.

No arquivo *new_layers.py* foram implementadas as camadas *FullyConnected3d*, *LayerNorm2d* e *LayerNorm3d*. As seguintes camadas foram construídas herdando os métodos e atributos da classe *Layer* do Joey, sem perdas para a aplicação. No arquivo *utils.py* estão localizados métodos auxiliares para instanciar *Function's* do Devito, servindo com o propósito de criação de tensores de dimensão 1, 2, 3, e 4, podendo estes tensores receber dimensões predefinidas ou iniciar novas.

As classes *MultiHeadAttention* e *VisionEncoder* foram dispostas no diretório *module* e a classe modelo *ViT* segue localizada na pasta *models*. Um arquivo de teste para a *VisionTransformer* foi incorporado ao diretório *tests*, contendo toda lógica aplicada para a fase *forward* do modelo, utilizando a base MNIST.

Todos os código construídos e testes implementados com a rede de forma geral, estão disponíveis no repositório criado a partir da derivação do código original do Joey[5]

.

[5] Disponível em <https://github.com/carloshnpa/joey>

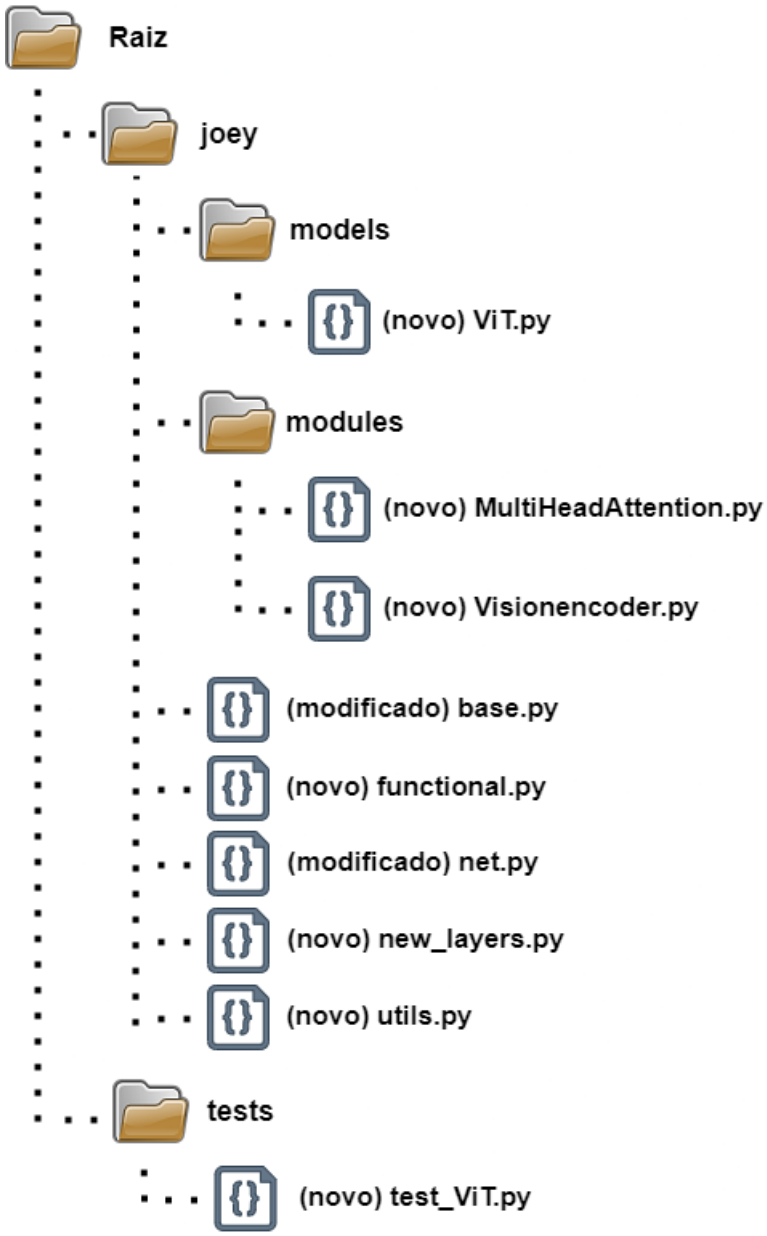


Figura 13 – Arquivos incluídos e modificados no Joey

Fonte: o autor

4 Resultados

Neste capítulo são apresentados os resultados obtidos na pesquisa de desenvolvimento e aprimoramento do atual framework Joey. Os Gráficos apresentados foram produzidos através da plataforma *Weights and Bias* e da biblioteca *matplotlib*, ambos disponíveis na linguagem Python.

4.1 Treinamento da rede

Para treinamento da rede, foi utilizado o método de transferência de aprendizagem, onde o modelo criado no PyTorch foi treinado e seus pesos e bias ajustados foram transferidos para a rede elaborada no Joey.

Para treinar a rede Transformer classificadora de imagens, a base de dados MNIST foi utilizada, contando cada *batch* de treinamento, 64 amostras de imagens de dígitos variando do 0 ao 9 de forma randômica. A base de dados foi iterada em 5 épocas de treinamento para garantir a maior acurácia possível.

A eficácia e tempo de processamento da rede foram medidos utilizando a plataforma do *Colab*, por oferecer um ambiente com configurações padrões sem beneficiar ou prejudicar qualquer comparação proposta. O resultado do treinamento pode ser conferido nas figuras 14, 15 e 16. O modelo alcançou durantes os testes eficiência em identificar corretamente 98% dos dígitos da base MNIST.

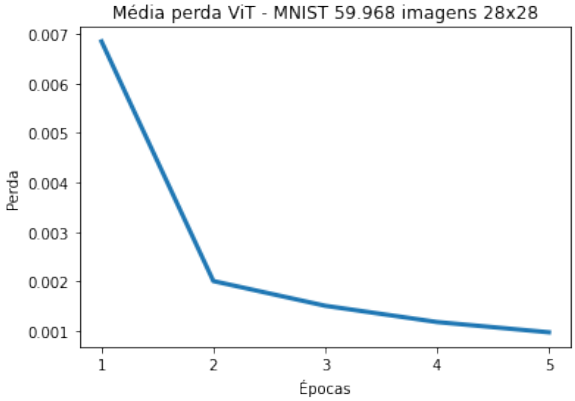
Após o treinamento da rede elaborada a partir do framewrok PyTorch, os pesos e bias de todas as camadas foram salvos utilizando uma função nativa da biblioteca, que permite recuperar os dados em uma rede de camadas equivalentes, como o caso da rede ViT elabora com o Joey. Os resultados são apresentados a seguir.

4.2 Tempo de processamento

Com a transferência de pesos e bias aplicada ao modelo ViT do Joey, testes comparativos entre a rede com o seu clone baseado no PyTorch foram realizados. Cada modelo executou na mesma instância de máquina alocada no Colab, porém em momentos distintos, para que cada rede fornecesse um detalhamento isolado na sua execução em termos de tempo de processamento, consumo de memória e utilização de CPU.

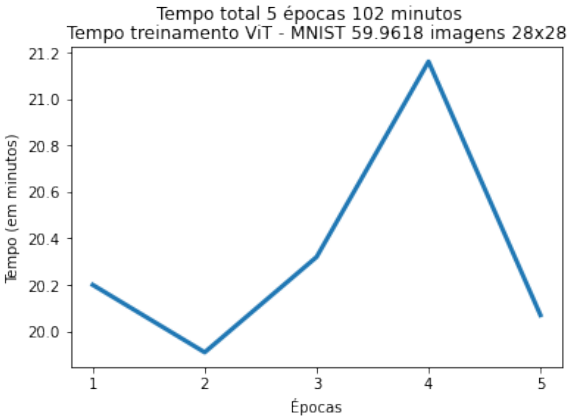
A base de validação foi a mesma MNIST, porém na proporção de testes, utilizando um total de 29.760 imagens. O tempo total requerido pela rede elabora no framework foi de 1 minuto e 22 segundos, comparado a 5 minutos e 50 segundo requeridos pela rede ViT

Figura 14 – Média de perda do treinamento da rede ViT - PyTorch



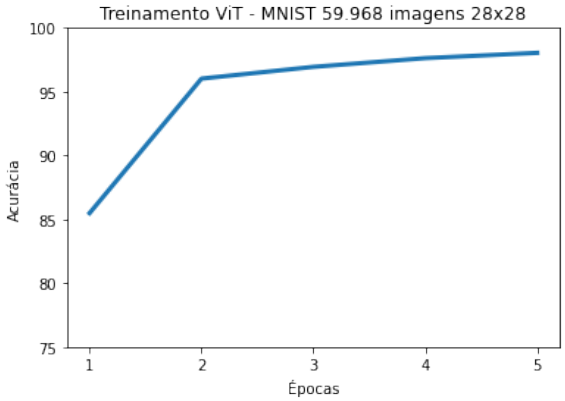
Fonte: o autor

Figura 15 – Tempo de processamento em cada época da rede ViT - PyTorch



Fonte: o autor

Figura 16 – Acurácia adquirida ao final de cada época da rede ViT - PyTorch



Fonte: o autor

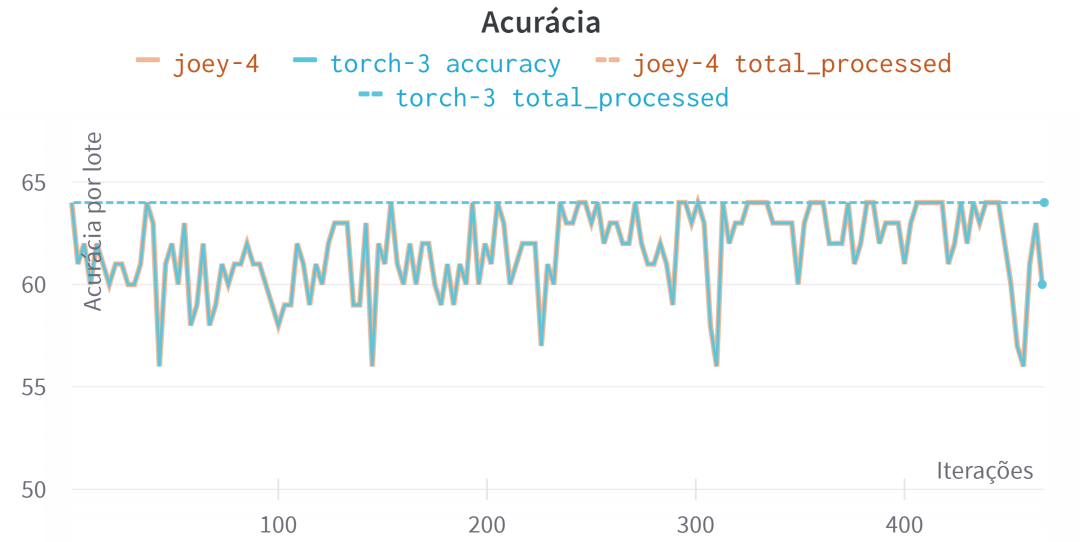
no Joey. A média de tempo necessária para processamento do lote de 64 imagens pelo framework PyTorch foi de 0.49 segundos, comparado a média de 2.23 segundos utilizando o framework Joey. O resultado do tempo de processamento é mostrado na figura 18

Como os pesos e bias de ambas as redes foram iguais, a acurácia produzida pelos modelos foi 96.36%, confirmando que a implementação das novas camadas foi feita com sucesso e a construção da rede foi equivalente para o modelo *ViT* no Joey. A acurácia dos modelos pode ser conferida na figura 17, onde o nível 64 no gráfico indica 100% de acerto para a amostra oferecida.

4.3 Consumo de recursos

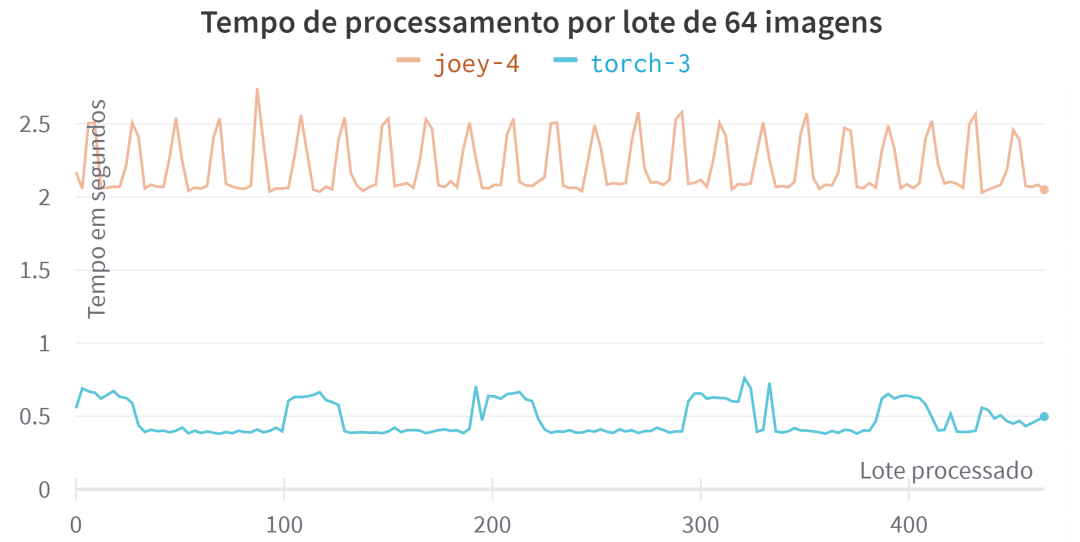
Os resultados de consumo de recursos foram aplicados a 4 métricas:

Figura 17 – Acurácia por lote - PyTorch vs. Joey



Fonte: o autor

Figura 18 – Tempo de processamento por lote - PyTorch vs. Joey



Fonte: o autor

- Utilização de CPU Total
- Utilização de CPU por núcleo
- Uso de threads
- Uso de memória

Em relação a utilização da CPU total (Figura 19), o consumo de ambos os fra-

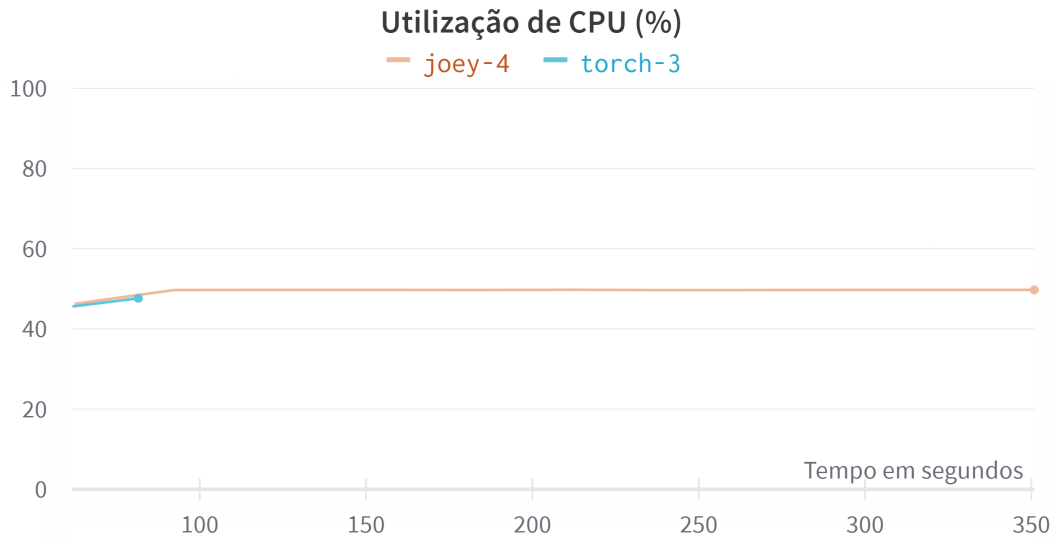
meworks foi parecido, porém, como o tempo de processamento do Joey foi maior, a utilização de CPU se manteve ao longo de todo o processo de validação.

Analisando a segmentação dos núcleos da CPU em cada modelo (Figura 20), a utilização dos núcleos 0 e 1 no PyTorch é ligeiramente maior que no Joey, porém na validação dos últimos lotes no modelo Joey, o uso do núcleo 0 supera o uso computado para o modelo em PyTorch.

Durante a fase de validação, a rede PyTorch manteve contínuo a utilização de 23 processos em paralelos, contra 19 processos iniciais observados na rede Joey, seguido do aumento de 1 processo em paralelo, como é ilustrado na Figura 21.

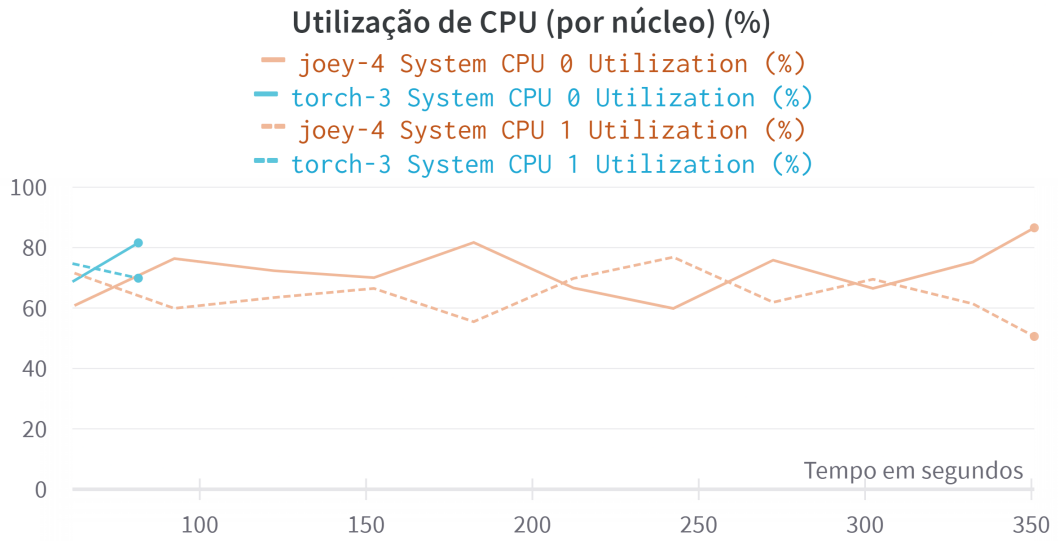
A média percentual de utilização de memória para o modelo produzido pelo framewrok PyTorch foi de 16.60%, comparado a média de 12.00% de utilização do modelo produzido pelo Joey.

Figura 19 – Utilização de CPU - PyTorch vs. Joey



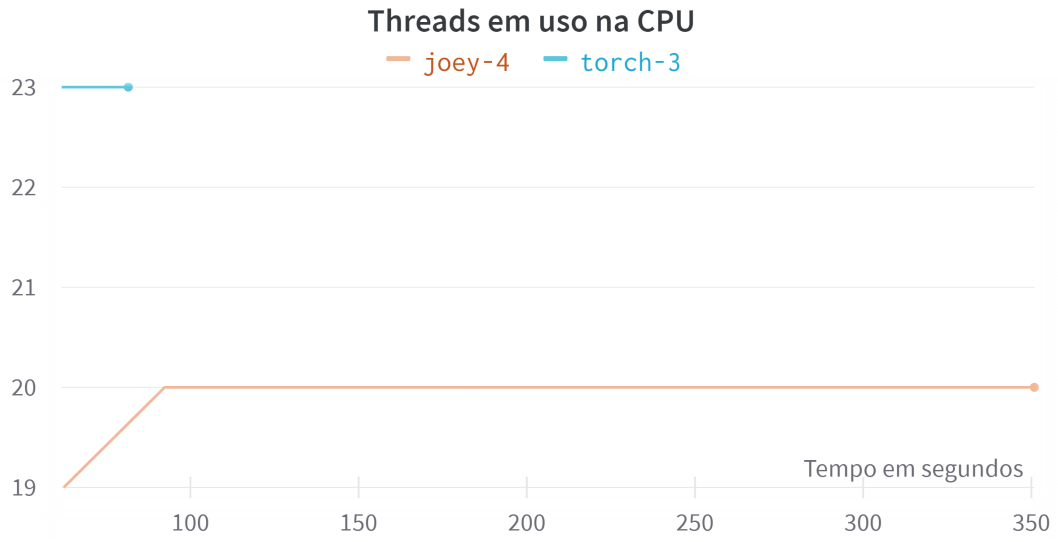
Fonte: o autor

Figura 20 – Utilização de CPU por núcleo - PyTorch vs. Joey



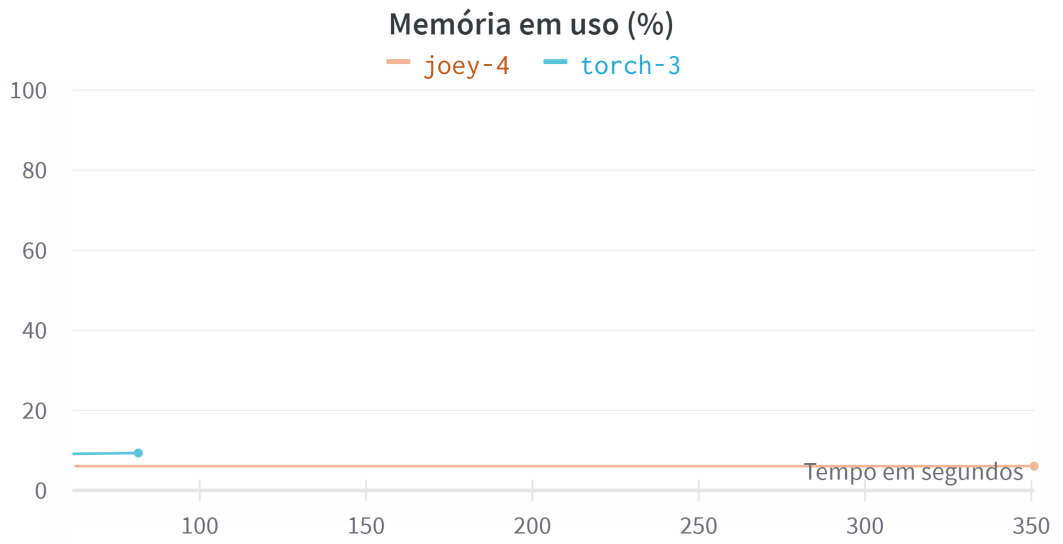
Fonte: o autor

Figura 21 – Uso de Threads - PyTorch vs. Joey



Fonte: o autor

Figura 22 – Utilização de Memória - PyTorch vs. Joey



Fonte: o autor

5 Conclusão

Neste capítulo são apresentados as conclusões sobre a pesquisa de desenvolvimento, limitações, considerações finais e propostas de trabalhos futuros.

5.1 Limitações

A grande limitação encontrada neste trabalho foi a submissão do modelo para treinamento utilizando apenas o framework Joey, sem a dependência de pré-treinamentos externos e transferência de pesos e bias. Com a criação das novas camadas, as equações de *backpropagation* não foram implementadas, impossibilitando o seu treinamento efetivo e isolado.

A fase de treinamento utilizando o método de *backpropagation*, utiliza dos conceitos de derivação das equações de cada camada, junto com a propagação reversa do erro para que o modelo possa ser treinado. Como a rede realiza uma operação de redução de tamanho nos tensores, não foi possível encontrar na literatura quais operações devem ser aplicadas neste caso, inviabilizando o desenvolvimento completo do treinamento da rede.

Por ser um framework ainda em construção, o Joey não possui uma documentação formal, se limitando aos exemplos aplicados, deixando a cargo do usuário o entendimento de como as camadas funcionam e se comunicam.

Para extrair ao máximo as otimizações oferecidas pelo Devito, a configuração de melhores variáveis de compilação se torna um passo importante, podendo reduzir em muito o custo computacional. Por padrão a classe *Operator* realiza as otimizações e geração de códigos sem tomar como referência as características mais profundas da máquina, como a presença de placa de processamento gráfico GPU, qual o modelo do processador e qual sua arquitetura. A customização dos otimizadores é exemplificada no repositório oficial do Devito, mas sem aprofundamento de como e quando elas devem ser aplicadas, deixando estas otimizações de forma empírica.

5.2 Considerações finais

Mesmo com resultados de tempo de processamento superiores ao framework PyTorch, o framewrok Joey ainda pode ser melhorado e otimizado, podendo inclusive superar o seu rival nos testes aqui propostos. A curva de aprendizado do Devito é simples, uma vez que o usuário entenda como as equações e dimensões funcionam neste framework, fica claro como se da a implementação das operações matriciais.

5.3 Trabalhos futuros

Apesar das novas implementações propostas para o framework Joey, melhorias como a implementação das equações *backward* podem ser aplicadas, garantindo assim que a rede possa ser treinada em sua totalidade, sem a necessidade de transferência de pesos e bias ajustados.

A criação de novas funções de ativação e novas camadas também são de grande valia para o Joey, possibilitando que o framework ganhe espaço na construção de novas redes, se tornando assim um framework ainda mais completo.

A elaboração de uma documentação formal para o framework Joey, traria maior clareza aos usuários dessa aplicação sobre quais passos devem ser tomados para a elaboração de uma rede neural.

Os otimizadores do Devito precisam de configurações manuais, estas, não apresentadas no presente trabalho. Explorar mais a fundo como as otimizações podem ser aplicadas em cada camada, pode resultar em uma saída ainda mais rápida e eficiente no consumo de recursos computacionais.

Transformers originalmente, são propostas para problemas de processamento de linguagem natural, a implementação de novos módulos para o framework, como a camada *Decoder* e ajustes para a construção do modelo para este problema, são de grande valia.

Testes envolvendo a distribuição em computação paralela com o MPI podem mostrar resultados ainda mais valiosos, elevando a classe do framework atual.

Referências

BA, J. L.; KIROS, J. R.; HINTON, G. E. *Layer Normalization*. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1607.06450>. Citado na página 36.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, v. 5, n. 2, p. 157–166, 1994. Citado na página 13.

BROWN, T. et al. Language models are few-shot learners. In: LAROCHELLE, H. et al. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2020. v. 33, p. 1877–1901. Disponível em: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf>. Citado 3 vezes nas páginas 13, 20 e 22.

CHATZITHEOKLITOS. *Joey, a new machine learning framework*. [S.l.]: Imperial College London, 2020. Citado 5 vezes nas páginas 13, 29, 30, 31 e 33.

DEVLIN, J. et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv, 2018. Disponível em: <https://arxiv.org/abs/1810.04805>. Citado 3 vezes nas páginas 13, 20 e 23.

DOSOVITSKIY, A. et al. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. Disponível em: <https://arxiv.org/abs/2010.11929>. Citado 3 vezes nas páginas 23, 24 e 35.

GOODFELLOW, I. J.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. <http://www.deeplearningbook.org>. Citado 6 vezes nas páginas 13, 17, 18, 19, 25 e 38.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural Comput.*, MIT Press, Cambridge, MA, USA, v. 9, n. 8, p. 1735–1780, nov 1997. ISSN 0899-7667. Disponível em: <https://doi.org/10.1162/neco.1997.9.8.1735>. Citado na página 18.

KUKREJA, N. et al. Devito: automated fast finite difference computation. *CoRR*, abs/1608.08658, 2016. Disponível em: <http://arxiv.org/abs/1608.08658>. Citado na página 13.

LECUN, Y. et al. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, v. 1, n. 4, p. 541–551, 12 1989. ISSN 0899-7667. Disponível em: <https://doi.org/10.1162/neco.1989.1.4.541>. Citado 4 vezes nas páginas 13, 18, 29 e 33.

LONDON mperial C. *Devito: Symbolic Finite Difference Computation*. 2016. Disponível em: <https://www.devitoproject.org/>. Citado na página 27.

LUPORINI, F. et al. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, Association for Computing Machinery, New York, NY, USA, v. 46, n. 1, apr 2020. ISSN 0098-3500. Disponível em: <https://doi-org.ez28.periodicos.capes.gov.br/10.1145/3374916>. Citado 2 vezes nas páginas 27 e 28.

PASZKE, A. et al. *PyTorch*. [S.l.]: GitHub, 2019. <https://github.com/pytorch/pytorch>. Citado na página 15.

SIMONYAN, K.; ZISSERMAN, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. Citado na página 33.

STERLING, T.; ANDERSON, M.; BRODOWICZ, M. *High Performance Computing: Modern Systems and Practices*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 012420158X. Citado 2 vezes nas páginas 25 e 26.

TEAM, N. D. *NumPy: A fundamental package for scientific computing with Python*. 2021. Disponível em: <https://numpy.org/doc/stable/reference/generated/numpy.einsum.html>. Acesso em: 28 fev 2023. Citado 2 vezes nas páginas 36 e 38.

VASWANI, A. et al. Attention is all you need. *CoRR*, abs/1706.03762, 2017. Disponível em: <http://arxiv.org/abs/1706.03762>. Citado 6 vezes nas páginas 13, 17, 19, 20, 21 e 22.