UMA ESTRUTURA PARA EXECUÇÃO DE REDES NEURAIS EVOLUTIVAS NA GPU

Jorge Rama Krsna Mandoju

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Geraldo Zimbrão da Silva

Rio de Janeiro
Setembro de 2019

UMA ESTRUTURA PARA EXECUÇÃO DE REDES NEURAIS EVOLUTIVAS NA GPU

Jorge Rama Krsna Mandoju

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Geraldo Zimbrão da Silva, D.Sc.

Prof. Geraldo Bonorino Xexéo, D.Sc.

Prof. Leandro Guimarães Marques Alvim, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
SETEMBRO DE 2019

Mandoju, Jorge Rama Krsna

Uma estrutura para execução de redes neurais evolutivas na GPU/Jorge Rama Krsna Mandoju. – Rio de Janeiro: UFRJ/COPPE, 2019.

XII, 63 p.: il.; 29, 7cm.

Orientador: Geraldo Zimbrão da Silva

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2019.

Referências Bibliográficas: p. 56 – 61.

1. Redes Neurais Evolutivas. 2. GPU. 3. CUDA. I. Silva, Geraldo Zimbrão da. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

O ignorante afirma, o sábio
duvida, o sensato reflete.
Aristóteles

Agradecimentos

Agradeço em primeiro lugar a Deus, que me ofertou a vida, pois sem ela eu não teria a oportunidade de presenciar e participar deste enorme universo no qual vivemos.

Agradeço ao meu pai que me ofereceu todo o suporte afim de que eu conseguisse alcançar os meus objetivos, estando sempre presente, lutando sempre para me oferecer um futuro melhor na busca da realização dos meus sonhos.

Agradeço a minha mãe que me ofereceu toda minha base ética e moral. Que me apoiou nos meus momentos de dificuldade em toda esta trajetória, pelo seu amor incondicional, cuidando de mim em todos os momentos

Agradeço ao meu irmão Júlio Rama, que me presenteou muitos risos e apoio nos momentos difíceis.

Aproveito para agradecer ao meu orientador Doutor Geraldo Zimbrão Da Silva que com sua sabedoria e paciência me encorajou a me lançar nesta jornada na qual estou convicto de ter sido bem assistido devido a sua grande experiência nesta área.

Aos meus amigos Vitor Machado, Paulo Xavier, Leonardo Gonçalves, André Brito quero expressar minha gratidão pela amizade, brincadeiras e incentivo que foram fundamentais para elevar o meu ânimo e descontrair nos momentos tensos. Sem eles, tudo teria ficado mais difícil. E a todos os outros amigos do laboratório ou não aqui não citados porém não menos importante neste processo, deixo registrado o meu agradecimento.

Agradeço a fundação COPPETEC pela oportunidade de trabalho e estudo.

Não posso deixar de agradecer a CAPES, pois o presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

E a todos aqueles que de forma direta ou indireta contribuíram para realização deste sonho, quero expressar a minha mais sincera gratidão.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UMA ESTRUTURA PARA EXECUÇÃO DE REDES NEURAIS EVOLUTIVAS NA GPU

Jorge Rama Krsna Mandoju

Setembro/2019

Orientador: Geraldo Zimbrão da Silva

Programa: Engenharia de Sistemas e Computação

Em neuroevolução, redes neurais são treinadas utilizando algoritmos evolutivos ao invés de utilizar o método do gradiente descendente. Uma das vantagens em relação ao método do gradiente descendente, é que torna possível além de definir o valor dos pesos de uma rede neural, também sua estrutura. Na otimização de redes neurais evolutivas com mesmo peso, são avaliadas todas as redes neurais de uma população para verificar qual é o valor da função *fitness* que cada rede neural irá possuir e com este valor, verificar quais são as redes neurais que irão passar para a próxima geração. O *GPU* (Graphic Processor Unit) é bastante utilizado nos treinos de redes neurais, devido a sua alta capacidade de paralelismo [1]. Porém, devido a sua arquitetura ser diferente de um processador comum, alguns algoritmos precisam ser executados de maneira diferente para aproveitar o aumento de desempenho que a arquitetura pode oferecer. Neste trabalho é criada uma arquitetura que seja capaz de diminuir o tempo de treino das redes neurais evolutivas através da junção dos pesos de toda população por camada fazendo com que cada camada represente os pesos de toda população. Desta forma é possível vetorizar as funções de avaliação de redes neurais. No treino para classificar o dataset MNIST, esta estrutura conseguiu obter um ganho de desempenho de até 64% nas redes neurais *MLP* e um speedup de 20 no cálculo do fitness.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A STRUCTURE FOR EVOLUTIONARY NEURAL NETWORKS IN GPU

Jorge Rama Krsna Mandoju

September/2019

Advisor: Geraldo Zimbrão da Silva

Department: Systems Engineering and Computer Science

In neuroevolution, neural networks are trained using evolutionary algorithms instead of the gradient descent method. One of the advantages over the gradient descent method is that it makes it possible not only to define the value of the weights of a neural network, but also its structure. In the optimization of evolutionary neural networks with the same weight, all neural networks of a population are evaluated to verify what is the value of the fitness function that each neural network will possess and with this value, to verify which are the neural networks that will move on to the next generation. The GPU (Graphic Processor Unit) is widely used in neural network training due to its high parallelism capability [1]. However, because their architecture is different from a common processor, some algorithms need to be executed differently to take advantage of the increased performance that the architecture can provide. In this work an architecture is created that is able to reduce the training time of evolutionary neural networks by joining the weights of all population by layer making each layer represent the weights of the entire population. In this way it is possible to vectorize the evaluation functions of neural networks. In training to classify the MNIST dataset, this structure has achieved a performance gain of up to 64% in neural networks MLP and a speedup of 20 in fitness calculation.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este capítulo é composto por uma breve introdução, em seguida apresentamos a motivação e os principais objetivos abordados em nosso trabalho. O último tópico trata da definição da estrutura de leitura a fim de facilitar a compreensão do trabalho.

1.1 Motivação

Em *machine learning*, o objetivo é descobrir ou inferir algum padrão a partir de um conjunto de dados. Existem dois conjuntos de dados: o de treino e o de teste. O algoritmo de *machine learning* precisa ser treinado utilizando os dados do conjunto de treino e, após o algoritmo estar devidamente treinado, ele irá prever e validar o resultado do conjunto de testes [2].

Atualmente para treinar uma rede neural, o método mais utilizado para problemas de classificação é o *backpropagation* [3]. O motivo pelo qual o *backpropagation* tem sido utilizado para treino é devido aos resultados bastante eficazes [4] na otimização dos pesos. Uma das técnicas de *backpropagation* utilizada para otimizar os pesos é o gradiente descendente [5], o qual é utilizado pelo otimizador ADAM [6].

Além da utilização do backpropagation também é possível utilizar algoritmos evolutivos para treinar os pesos ou até mesmo a estrutura da rede [7, 8].

No caso de utilizar algoritmos evolutivos para treinar uma rede neural, foram obtidos casos de sucesso no treino dos pesos das redes neurais onde o ambiente para treino era complexo ou dinâmico, como por exemplo jogar um jogo virtual [9]. Também foram utilizados algoritmos evolutivos para definir qual seria a melhor estrutura que a rede neural iria possuir, porém o *backpropagation* continua a ser utilizado para otimizar os pesos da rede [10].

Um dos algoritmos evolutivos mais utilizados é o algoritmo genético, que é um algoritmo de otimização baseado em técnicas evolutivas apresentadas na biologia, como a hereditariedade dos genes, a mutação e a seleção natural [11]. Uma das vantagens deste algoritmo é que ele é facilmente paralelizável e escalável.

A GPU(*Graphic Processor Unit*) é um processador gráfico o qual oferece um grau de paralelismo maior que um processador comum [1]. Para utilizar este paralelismo, a Nvidia criou a arquitetura CUDA, que é uma arquitetura computacional que visa facilitar o desenvolvimento de programas paralelos na GPU [12].

Uma das métricas utilizadas para comparar poder de processamento nas GPUs é o FLOPs, que significa operações de ponto flutuante por segundo (*Float point operations per second*). Estudos mostrados pela Nvidia [13] demonstraram que com a utilização de uma GPU era possível obter teoricamente até 7 vezes mais FLOPs que o processador comum em operações de precisão única comparando a GTX 780 ti contra a arquitetura Ivy Bridge e até 2 vezes mais em operações de precisão dupla comparando com o Tesla K40 e arquitetura Ivy Bdrige. O cálculo da quantidade de FLOPs teórico utiliza 4 valores: a velocidade do *clock* processador, o número de cores que o processador possui, o número de instruções executadas por ciclo e número de CPUs por nó [14].

Apesar dos sucessos obtidos no processamento, não foi encontrado nenhum estudo sobre a utilização de GPUs na otimização dos pesos das redes neurais utilizando algoritmos evolutivos. Apenas foram encontrados artigos que utilizam a paralelização em nível de processos [15]. Entretanto, de acordo com alguns estudos é possível aumentar o desempenho da execução através da utilização de GPU no algoritmo genético [16–18]. Segundo POSPICHAL, JAROS e SCHWARZ, foi possível obter um aumento de desempenho de até sete mil vezes sobre a CPU convencional [16].

Trabalhar com a GPU impõe uma série de restrições, como por exemplo a memória, porque não é possível expandi-la e a inserção de uma outra GPU na mesma máquina apenas aumentaria o número de *threads* e não a expansão da memória em si [19]. Outra dificuldade apresentada é a necessidade de uma mudança no paradigma de programação para aproveitar o alto grau de paralelismo da arquitetura que a GPU oferece, pois devido à arquitetura CUDA (*Compute Unified Device Architecture*), o alto paralelismo é obtido por processadores que executam apenas a mesma instrução [13].

Atualmente existem ferramentas que facilitam a paralelização de processos de machine learning através de *workflow*. Uma das ferramentas mais famosas é o *Tensorflow* [20], a qual foi criada para a utilização de aprendizado de máquina. Como esta ferramenta possui um grau de abstração maior, não é possível manipular as operações em baixo nível dentro da GPU, exigindo que os algoritmos que são executados nela sejam bem estruturados para obter o maior grau de paralelismo.

Neste trabalho, foi criada uma estrutura que diminuiu o tempo de treino de redes neurais evolutivas *MLP* e convolucionais fazendo um uso maior da *GPU*. A estrutura ofereceu um ganho de desempenho de até 64% em relação ao tempo total

de execução nas redes *MLP*. Nas redes neurais convolucionais o ganho foi menor, pois o ganho do desempenho foi melhor observado na execução das camadas totalmente conectadas.

1.2 Objetivos

O presente trabalho possui os seguintes objetivos:

- Criação de uma estrutura que facilite a execução das redes neurais evolutivas na GPU de modo que seja possível obter um grau maior de utilização da GPU e diminuir o tempo de treino da mesma.
- Criação de um workflow paralelizável que execute o treino das redes neurais evolutivas na GPU, a fim de possibilitar a execução do treino das redes neurais evolutivas na GPU utilizando ferramentas mais modernas de paralelização.
- Implementar um algoritmo genético na GPU compatível com o workflow desenvolvido para este trabalho. Este algoritmo também deverá fazer o uso da estrutura proposta pelo trabalho, de modo que seja possível paralelizar operações genéticas como o *crossover* e a mutação.

1.3 Organização

Para uma melhor compreensão do trabalho, esta dissertação está organizada em cinco capítulos sendo este o primeiro.

O capítulo 2 será constituído de fundamentação teórica que aborda os principais temas que serão utilizados neste trabalho. A fundamentação iniciar-se-á abordando o algoritmo genético, em seguida as redes neurais e finaliza com as redes neurais evolutivas.

No capítulo 3 será abordado a proposta do trabalho, abordando o problema que será tratado e a proposta de solução para o mesmo.

No capítulo 4 será demonstrado a avaliação experimental. Neste capítulo avaliamos os objetivos dos experimentos, a metodologia , os resultados obtidos e suas análises.

No capítulo 5 será feita a conclusão do trabalho, apresentando as contribuições e os possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Neste capítulo iremos discorrer um pouco sobre o que é algoritmo genético e suas abordagens. O algoritmo genético será utilizado principalmente para treinar as redes neurais deste trabalho.

2.1 Redes Neurais

As Redes Neurais artificiais são um modelo computacional que possuem várias interconexões chamadas de neurônios. Este modelo foi inspirado nas redes neurais biológicas, onde cada neurônio gera entrada para outro [21].

2.1.1 Perceptron

O *Perceptron* [22] foi uma das primeiras redes neurais criadas. O modelo consiste em um neurônio que é conectado a múltiplas entradas, onde cada uma é multiplicada por um peso. O neurônio irá gerar uma saída como desativado ou ativado de acordo com a combinação das entradas recebidas. A equação resolvida por um perceptron é:

$$f(x) = \begin{cases} 0 & w * x + b \leq 0 \\ 1 & w * x + b > 0 \end{cases}$$

Onde:

- x é o valor da entrada que o *perceptron* recebe
- w é o valor do peso que irá multiplicar a entrada
- b é o bias que irá dizer o quanto o neurônio precisa para ativar.

A figura 2.1 para representar este perceptron:

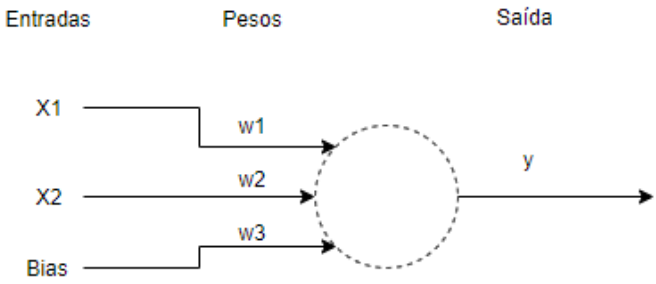

Figura 2.1: Exemplo de um perceptron

2.1.2 Redes Neurais Multicamadas (MLP)

As redes neurais de multicamadas (Multi Layer Perceptrons) [23] são redes neurais que possuem múltiplas camadas com *perceptrons* onde cada camada possui como entrada a saída dos *perceptrons* da camada anterior. A primeira camada recebe as entradas do problema, a última camada é chamada de camada de saída e as camadas que ficam entre a primeira e a última são chamadas de camada oculta. figura 2.2 demonstra um exemplo da estrutura:

Todos os neurônios, além de aplicar os pesos de acordo com os resultados das camadas anteriores, também aplicam uma função de ativação não linear. Uma das funções mais utilizadas é a função sigmoide. A função sigmoide pode ser descrita através da seguinte equação:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

Otimização para GPU

Uma maneira de otimizar a execução nestas redes neurais é estruturar os pesos de cada camada da rede neural como uma matriz 2D, onde uma dimensão irá representar os neurônios da camada anterior e a outra irá representar os neurônios da camada posterior. Logo iremos possuir um conjunto de matrizes de pesos, onde cada valor dentro desta matriz representa uma ligação de um neurônio específico de uma camada para outra camada. Com estas matrizes prontas, são executadas as seguintes etapas:

- Primeiro passo: É feita uma multiplicação de matriz da entrada com a matriz de pesos da primeira camada.
- Segundo passo: É executada a função de ativação em todos os valores gerados

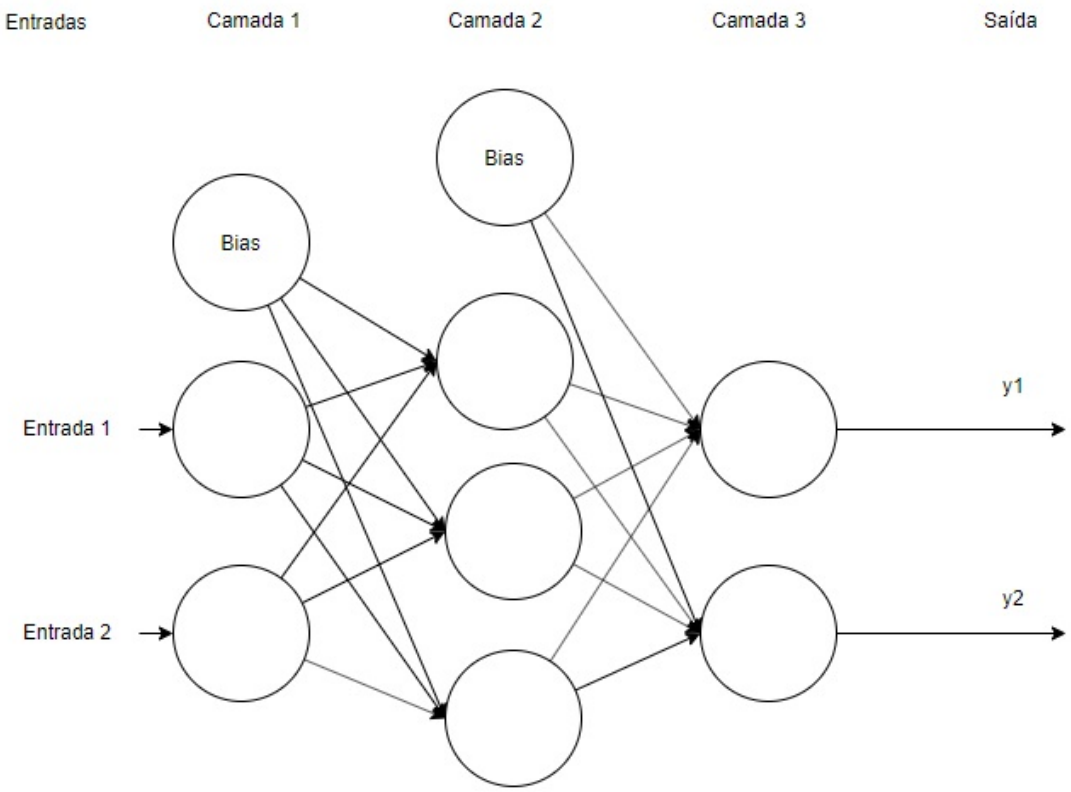

Figura 2.2: Exemplo de uma rede neural MLP

pelo passo anterior.

- Terceiro passo: Para cada camada restante:
  - É feito uma multiplicação da camada atual com o resultado da camada de entrada
  - É executada a função de ativação em todos os valores gerados pelo passo anterior

A vantagem da execução das redes neurais desta maneira é que a execução da multiplicação de matrizes na GPU é bastante eficiente [24], fazendo com que seja possível o maior grau de utilização na GPU. A figura 2.3 demonstra os passos executados.

2.1.3 Redes Neurais Convolucionais (CNN)

As redes neurais convolucionais (Convulational Neural Networks)[25] são redes neurais inspiradas no córtex visual biológico [26], as quais possuem múltiplas camadas onde os neurônios não são totalmente conectados com todos os neurônios da próxima camada. Elas consistem em uma camada de entrada e saída também, porém nas

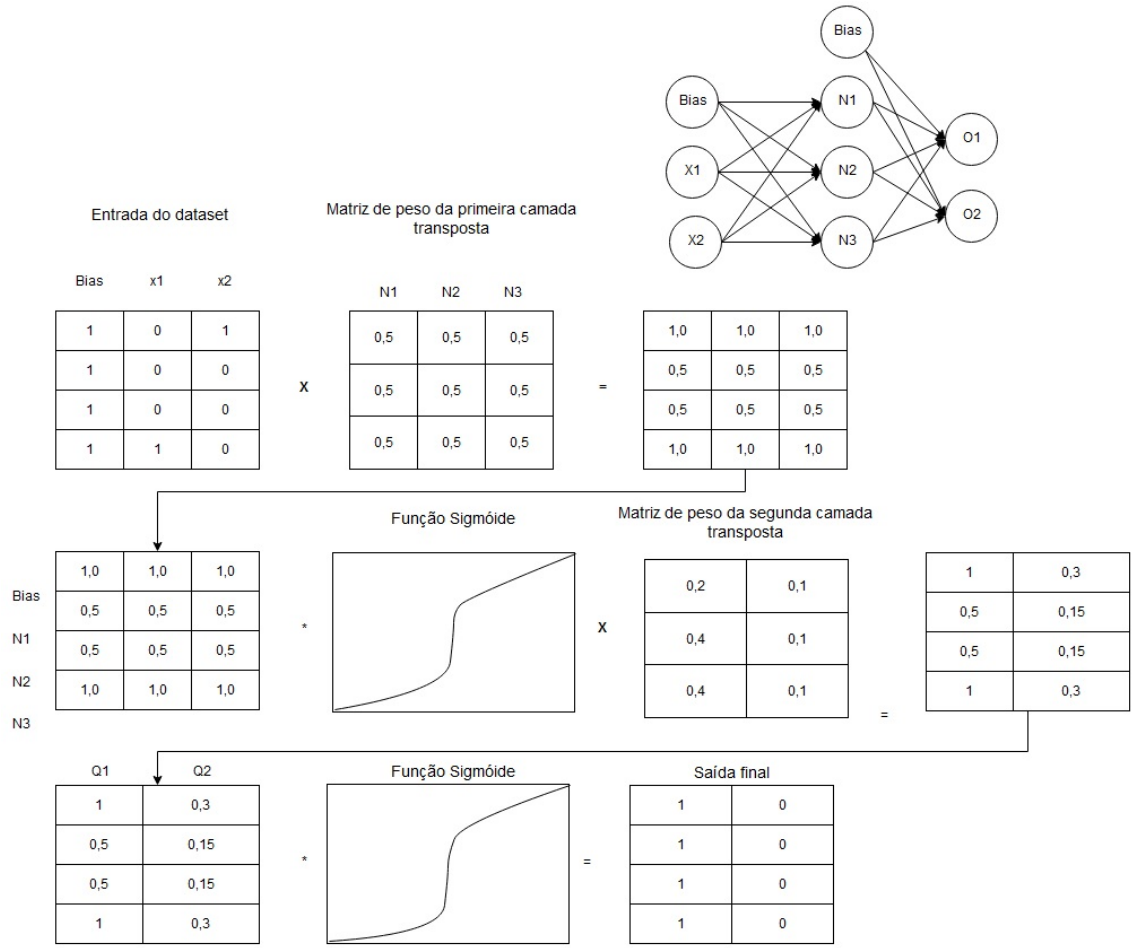

Figura 2.3: Exemplo da execução de uma rede neural multicamada

camadas ocultas elas possuem diversos tipos de camadas, podendo ser: camada convolucional, camada RELU, camada de agrupamento, camada totalmente conecta e camada de normalização. Um exemplo de uma Rede neural convolucional pode ser vista de acordo com a figura 2.4.

A seguir veremos uma revisão das operações executadas em cada camada.

Camada de convolução

Esta camada executa a operação de convolução, sendo esta uma operação que junta os dados de entrada com um filtro ou um *kernel* para produzir um mapa de funcionalidades. A operação é executada passando o filtro pela matriz de entrada. Para cada passagem do filtro, é executada uma multiplicação entre os valores da matriz de filtro e da entrada e em seguida é feita uma soma dos resultados. O filtro possui o parâmetro *stride*, que é o tamanho do passo que o filtro percorre pelos dados de entrada. É possível visualizar a execução da operação abaixo:

Assim como as redes neurais de múltiplas camadas, as redes neurais também possuem função de ativação. Após a execução da camada de convolução é executada

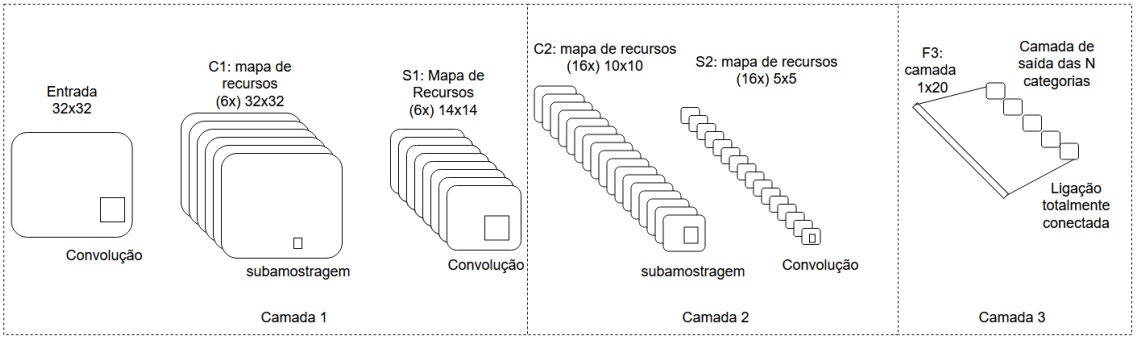

Figura 2.4: Exemplo da execução de uma rede neural convolucional

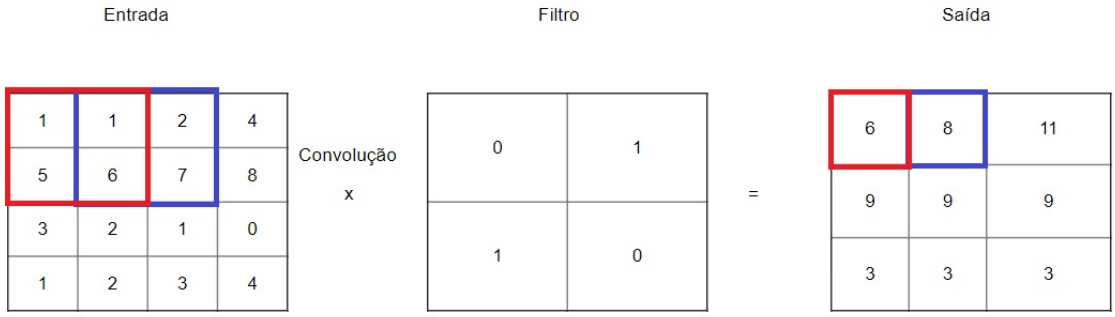

Figura 2.5: Exemplo da execução de uma convolução com filtro 2x2 e *stride* de tamanho 1

uma função de ativação. Uma das funções mais utilizadas para esta rede neural é a RELU [27]

A cada execução de uma convolução, os tamanho da saída diminui, fazendo com que seja necessário executar um preenchimento, o qual irá colocar camadas com os valores zero em volta da entrada, fazendo com que a saída da convolução não diminua.

Camada de agrupamento

A camada de agrupamento geralmente é executada depois de uma camada de convolução. A funcionalidade do agrupamento nas redes neurais convolucionais é diminuir o número de parâmetros na rede agrupando os valores.

Uma das funcionalidades mais utilizadas é o "agrupamento máximo", o qual divide as entradas em vários conjuntos contínuos e seleciona o valor máximo entre cada conjunto. O tamanho destes conjuntos é definido pela entrada do programa através do tamanho do filtro e *stride*. Um exemplo do agrupamento máximo pode ser observado na figura 2.6 :

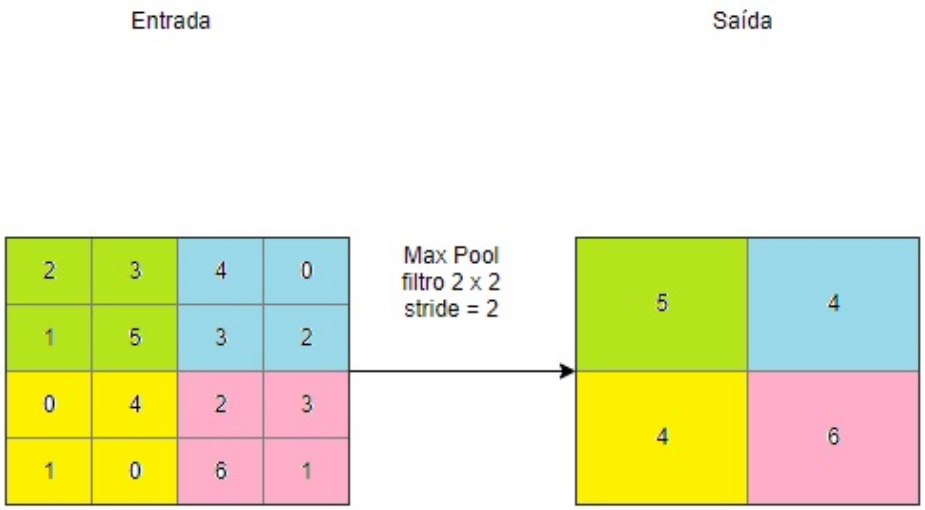

Figura 2.6: Exemplo da execução da camada de agrupamento utilizando filtro 2x2 e *stride* de tamanho 2

Camada totalmente conectada

Esta camada irá possuir uma rede neural de múltiplas camadas (Multi-Layer Perceptron). Nesta camada, diferentemente das outras, cada neurônio será conectado com todos os neurônios da camada posterior dentro da rede neural desta camada. Geralmente esta é a última camada executada pela rede neural convolucional, oferecendo o resultado da execução da rede por completo.

2.1.4 Gradiente descendente

O *gradiente descendente* é um algoritmo de otimização [5], que funciona de forma iterativa, onde tenta aproximar uma função para o mínimo local alterando os seus parâmetros a cada passo. Esta iteração pode ser definida pela equação abaixo:

$$w^{new} = w^{old} - \alpha \bigtriangledown wF(w)$$

onde:

- w^{new} representa o próximo valor dá iteração
- w^{old} representa o valor dá iteração
- α é a taxa de aprendizado

- $\bigtriangledown wF(w)$ é a direção é onde a função desce mais rápido

O critério de parada do algoritmo pode ser definido de duas maneiras, seja por número de iterações [28] ou utilizando a técnica de parada prematura (*early stopping*) [29] , onde o algoritmo finaliza após um número de épocas sem melhora.

O gradiente descendente também possui outras variantes que utilizam as características do gradiente para melhorar o resultado, como por exemplo o ADAM [6], o qual utiliza os gradientes quadráticos e o *momentum* para escalar a taxa de aprendizado, acelerando para o algoritmo chegar ao ponto de convergência mais rápido e também contribuindo para o algoritmo chegar em uma solução ótima melhor, diminuindo a taxa de aprendizado conforme chega mais perto do mínimo local.

2.2 Algoritmo Genético

Os algoritmos genéticos [30] são algoritmos de busca e otimização baseados na teoria de Charles Darwin de seleção natural e da genética de Mendel . A teoria de seleção natural é a teoria onde os indivíduos mais adaptados ao ambiente sobrevivem e cruzam entre si produzindo novas gerações.

Esses algoritmos possuem algumas características especiais que são diferentes em relação aos outros métodos de otimização e de busca:

- Eles trabalham fazendo uma codificação das possíveis soluções em vez de trabalhar com a solução como um todo.
- Usam uma população de soluções, avaliando várias soluções simultaneamente.
- Utilizam informações de custo e recompensa e não requerem derivadas de funções.
- Usam regras de transição probabilísticas e não determinísticas [30].

No algoritmo genético, cada possível solução gerada é considerado como um cromossomo. Cada cromossomo é descrito por um conjunto de gene. Estes cromossomos são avaliados por uma função *fitness*. O valor da função *fitness* define o quão perto o individuo pode estar da solução ótima, definindo o quão "adaptado"ele está na população em relação as outras soluções [11].

Cada iteração do algoritmo genético é considerada como uma geração. A cada geração, os indivíduos da população são selecionados baseados no seus respectivos *fitnesses*. Com base nesse valor, é utilizado um critério de seleção para os melhores indivíduos que irão permanecer para a próxima geração e também é aplicado operadores genéticos nos mesmos. Os indivíduos não selecionados para a próxima iteração

irão ser descartados. Existem vários critérios de seleção como o truncamento, torneio e roleta. Também existem vários operadores genéticos como o *crossosver* e a mutação.

Os algoritmos genéticos nem sempre oferecem a solução ótima, mas conseguem oferecer uma solução próxima do ótimo em um tempo hábil. Então, conforme visto, o fluxograma do algoritmo genético iria respeitar a figura 2.7:

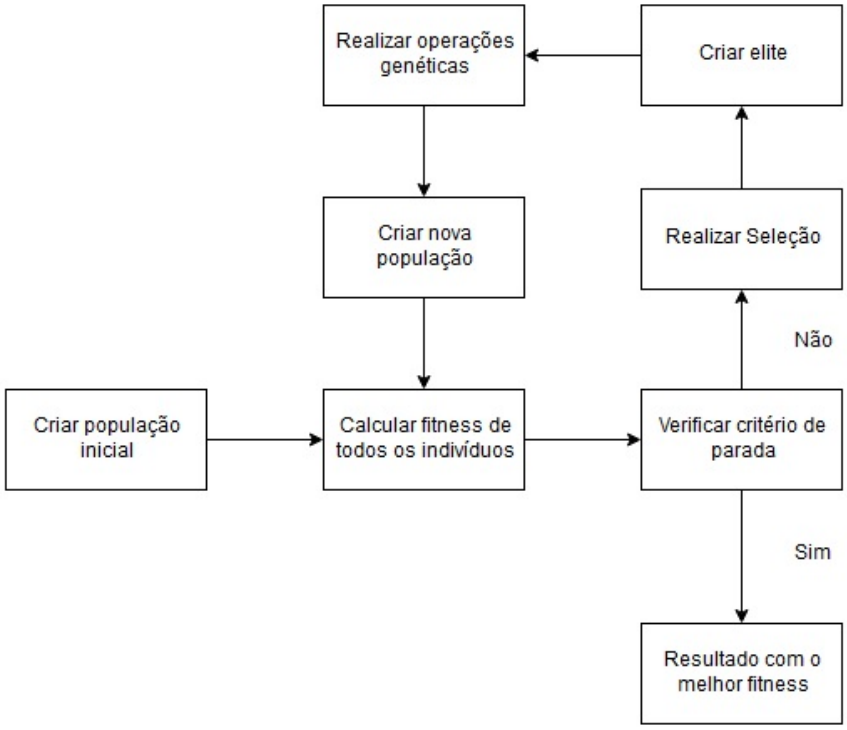

Figura 2.7: Esquema da execução do algoritmo genético

A partir do esquema acima descrito, apresentaremos com mais detalhe o algoritmo nas subseções que se seguem.

2.2.1 Definição dos Genes

Para a execução do algoritmo genético, primeiramente é necessário definir qual será a codificação dos genes das soluções. Um gene é uma fração do que seria a solução final. Esta é uma das definições mais importantes para o algoritmo genético, pois as soluções geradas pelo algoritmo serão manipulações em cima destes genes. No caso das redes neurais deste trabalho, um gene poderia ser facilmente um peso [31] ou até mesmo a estrutura da rede neural [32], e cada individuo seria representado por um conjunto destes genes, o qual é chamado de cromossoma.

2.2.2 Inicialização

Para iniciar o algoritmo genético, é necessário criar uma população inicial. Esta população é criada de maneira aleatória com genes diferentes. Esta maneira aleatória de gerar os genes deve garantir que pelo menos todas as possíveis soluções sejam possíveis de serem geradas [33]. Entretanto é possível iniciar a população iniciando com indivíduos considerados "bons"para facilitar e agilizar a manipulação dos mesmos [33].

2.2.3 Função *Fitness*

No algoritmo genético é necessário criar uma função que avalie o quão perto o individuo está da solução ideal. Esta função é chamada de *fitness* e é criada de acordo com o problema que o algoritmo genético irá resolver. Esta função avaliará todos os indivíduos dando uma pontuação para cada um deles e esta pontuação que será utilizada no passo de seleção.

2.2.4 Seleção

Para selecionar os indivíduos, é criado uma função que calcular o *fitness* de cada indivíduo. Esta função é um parâmetro importante para o algoritmo genético, pois irá definir o quão perto da solução cada indivíduo está.

Após calcular o *fitness* de cada indivíduo, cada indivíduo deverá ser selecionado baseado no critério de seleção. Com base nisso, o algoritmo genético entra como parâmetro o número de indivíduos que irão sofrer as operações genéticas [11]. Os indivíduos selecionados podem fazer parte de uma elite, mantendo para a próxima geração [34], ou não [35]. A seguir iremos ver alguns critérios de seleções utilizados.

Seleção por Truncamento

Nesta seleção, é criada uma lista ordenada com os fitnesses de cada indivíduo. Os indivíduos que possuírem os maiores fitnesses serão os indivíduos que irão para a próxima geração, enquanto todos os outros indivíduos morrem. O esquema desta seleção pode ser observado na figura 2.8:

Seleção por Roleta

A seleção por roleta é uma implementação da seleção proporcional [11]. Nessa seleção, os indivíduos são selecionados dentro de uma roleta, onde o tamanho da área que cada indivíduo ocupa na roleta é baseado no seu *fitness* [11]. Nesta seleção, a roleta é girada N vezes, onde N é o número de indivíduos que será selecionado para a próxima geração. A cada giro, é selecionado um individuo pela roleta. A

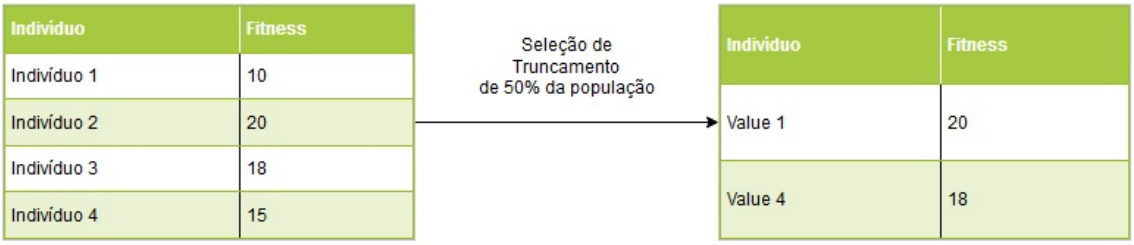

Figura 2.8: Exemplo de seleção de truncamento onde é selecionado 50% da população

probabilidade de um individuo qualquer ser selecionado é definida pela equação abaixo [36]:

$$\rho_i = \frac{f_i}{\sum_{n=1}^{M} f_n}$$

Onde:

- ρ_i é a probabilidade do individuo ser selecionado dentro da população
- f_i é o *fitness* que o individuo possui .
- $\sum_{n=1}^{M} f_n$ é a soma do fitness de todos os indivíduos dentro da população de tamanho M.

Este esquema de seleção pode ser visto pela figura 2.9:

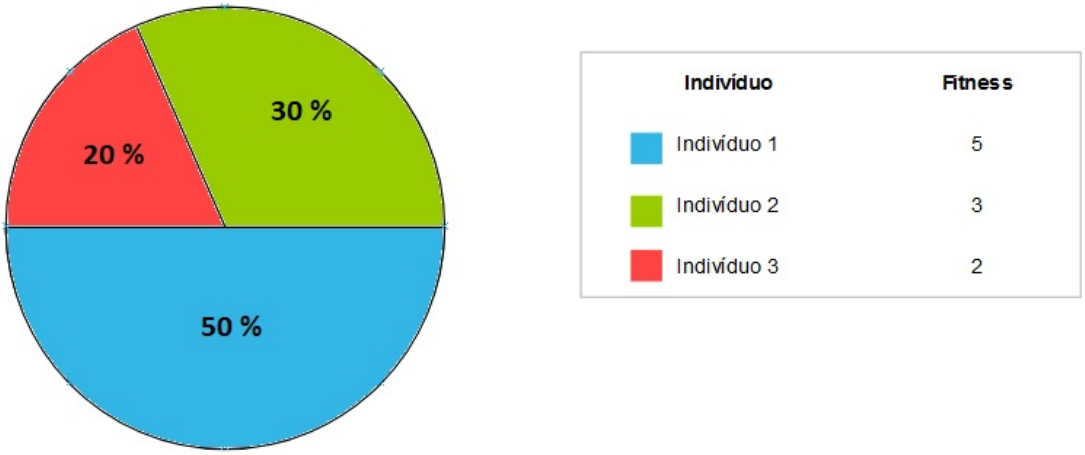

Figura 2.9: Exemplo de seleção de roleta com três indivíduos.

Devido esta seleção ser um método estocástico, existe uma pequena possibilidade de todos os indivíduos selecionados pela roleta sejam o pior indivíduo da geração [11]. Além disso, esta seleção possui um problema de convergência prematura, pois durante as seleções iniciais, o variância do fitness é alta na população e um pequeno

número de indivíduos é muito mais adaptados que os outros, fazendo com que apenas os indivíduos mais adaptados se reproduzam e se multipliquem na população, prevenindo o algoritmo genético de realizar uma busca mais profunda [11].

Seleção por Torneio

Neste tipo de seleção os indivíduos são separados em grupos com tamanho K e em seguida, são selecionados os indivíduos com maior *fitness* dentro de cada grupo. [11]. Uma das vantagens da seleção por torneio é diminuir o grau de convergência do algoritmo genético, fazendo com que indivíduos não tão bons, mas que possuam alguma característica boa sobrevivam por mais gerações, facilitando o encontro do máximo global. Este tipo de seleção consegue uma convergência mais rápida em relação a roleta [37], mas também gera uma convergência prematura mais rápida [38]. Um exemplo da seleção pode ser visto na figura 2.10.

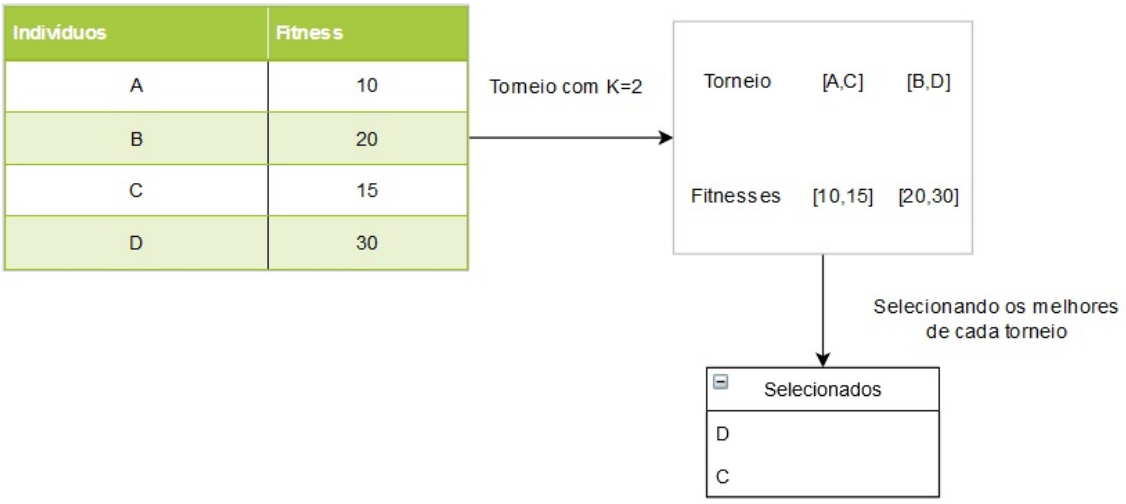

Figura 2.10: Exemplo de seleção de torneio, com torneios de tamanho 2.

Elitismo

No elitismo, Os indivíduos com maior fitness se mantém para a próxima geração e não sofrem operações genéticas. A vantagem desta seleção é acelerar a convergência do algoritmo [11].

2.2.5 Operadores Genéticos

Os operadores genéticos são funções que irão gerar novos indivíduos a partir dos indivíduos selecionados pela etapa de seleção. Podemos dividir os operadores genéticos em 2 subgrupos: o *crossover* e a mutação. A eficiência dos operadores genéticos depende da codificação dos genes [11].

Crossover

O *crossover* é um operador genético que utiliza dois indivíduos, cruzando os genes entre si. Os indivíduos que irão receber o operador são chamados de pais [11]. Os indivíduos são selecionados aleatoriamente. Abordaremos três tipos de crossover visto em [39]:

- Crossover de um ponto
- Crossover multiponto
- Crossover uniforme

No crossover de um único ponto, é escolhido um valor aleatório de 0 a L, onde L é o tamanho do cromossomo. Este ponto escolhido será o ponto de cruzamento, onde os valores localizados em um lado em relação ao ponto de cruzamento serão os genes de um dos pais e o outro lado serão os genes do outro pai. A figura 2.11 demonstra o funcionamento deste operador:

Figura 2.11: Exemplo de crossover de um único ponto

No *Crossover* multiponto, são selecionados dois pontos com valores aleatórios de 0 a L, onde L sendo o tamanho da somas dos cromossomo. A partir destes dois pontos é gerado um intervalo entre esses dois pontos. Com o intervalo gerado, o filho irá herdar de um dos pais os genes encontrados dentro deste intervalo e do outro fora do intervalo selecionado. A figura 2.12 demonstra o funcionamento deste operador:

No *Crossover* uniforme, cada gene é trocado entre os pais com uma probabilidade p, que definirá de qual pai o gene irá ser herdado. Geralmente esta probabilidade é 0.5 para manter a simetria entre genes passados pelos pais [39]. A partir desta probabilidade, é escolhido de qual pai será herdado cada gene. Uma maneira de representar de qual parente que cada gene irá receber é através de uma máscara

Figura 2.12: Exemplo de crossover de multiponto

binária, que caso seja 0 irá herdar o valor deu um dos pais e 1 do outro pai [40]. A figura 2.13 representa o funcionamento deste *crossover*:

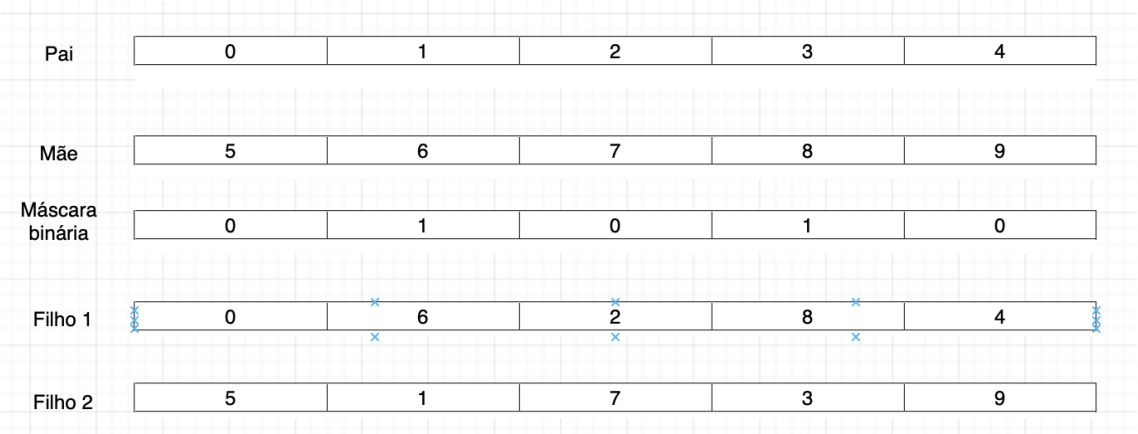

Figura 2.13: Exemplo de crossover uniforme

Mutação

A mutação é um operador genético que trabalha sobre um único indivíduo. Este operador garante que a população não fique permanentemente em um locus particular (posição de um gene) [11] e também que o algoritmo não se prenda no ótimo local. Outra vantagem deste operador é que ele possui um efeito pequeno no fenótipo [36].

Existem várias maneiras de implementar uma mutação, e elas dependem da codificação do genes implementada no algoritmo genético. Um exemplo básico pode ser verificado na figura 2.14 onde cada gene tem uma probabilidade para ser selecionado pelo operador de mutação, e os genes selecionados terão o seu valor alterado por um número aleatório. A figura 2.14 demonstra um exemplo de uma mutação aplicada num *array* de genes binário.

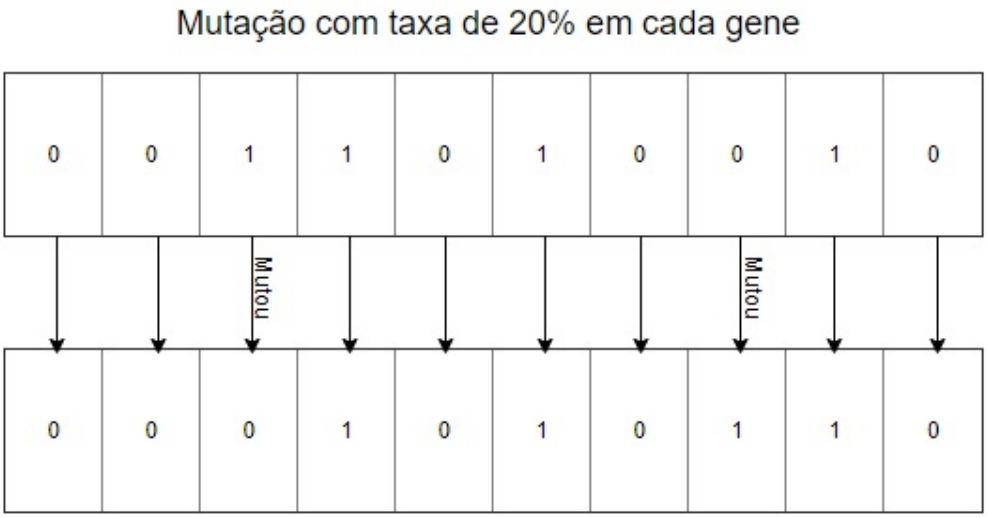

Figura 2.14: Exemplo de mutação em um *array* binário

2.2.6 Principais parâmetros

Os parâmetros do algoritmo genético são bastante importantes para definir a execução do mesmo. De acordo com [41], os principais parâmetros são:

- Tamanho da população: Este parâmetro define o tamanho que a população irá possuir durante a execução do algoritmo. O tamanho da população tem uma grande influencia no desempenho e resultado do algoritmo genético. Com uma população muito pequena, o algoritmo genético irá trabalhar em um espaço de busca muito pequeno, fazendo com que não convirja para um bom resultado. Aumentando o tamanho da população, irá aumentar o tamanho do espaço de busca percorrido pelo algoritmo, aumentando a probabilidade do algoritmo convergir para um resultado bom, porém, isso irá exigir mais poder computacional, devido ao maior número de indivíduos que irão ser postos na memória e o maior número de cálculos que irá ser necessário utilizar em cada individuo (como calcular o fitness por exemplo).

- Taxa de *crossover*: Este parâmetro define a taxa de *crossover* que irá possuir a população. Quanto maior a taxa, mais indivíduos novos serão gerados com base nos melhores indivíduos da população, fazendo com que a população convirja mais rápido para uma solução. Porém, convergir muito rápido para uma solução não necessariamente irá ser uma solução ótima, pois não irá reduzir drasticamente o espaço de busca. O valor deste parâmetro não pode também ser muito baixo pois a busca irá ficar estagnada e não irá convergir para um bom resultado.

- Taxa de mutação: Este parâmetro define a taxa de *mutação* que a população irá possuir. Ela irá aumentar a diversidade genética, aumentando o espaço de busca do algoritmo. Taxas muitos elevadas torna a busca completamente aleatória.

2.3 Redes Neurais Evolutivas

As redes neurais evolutivas são redes neurais que utilizam algoritmos evolutivos (como o algoritmo genético) como otimizador. Existem várias maneiras de otimizar uma rede neural através de algoritmos evolutivos, seja apenas pelos pesos [31], ou pela estrutura [42], ou os dois ao mesmo tempo [8]. É possível também otimizar as regras de aprendizado, definindo os parâmetros que o algoritmo genético irá possuir de acordo com a evolução da rede neural [31].

Para otimização, dependendo do quê e como se quer ser otimizado, é necessária uma codificação para os genes.

Existem duas maneiras de codificar os genes para um rede neural evolutiva: codificação direta e codificação indireta [43] . Na codificação direta os genes representam a rede neural em si, seja representando os pesos ou a estrutura da rede neural como ela é. A vantagem desta codificação é que não é necessário fazer execução de passos a mais para a execução das redes neurais no cálculo dos fitnesses. Na codificação indireta, os genes representam funções que irão gerar a estrutura ou rede neural em si. A codificação indireta possui algumas vantagens como o uso menor da memória pois neste caso os genes irão representar estruturas menores em relação a uma estrutura da rede neural completa e mapear os genes de maneira que se adéquem ao problema proposto, fazendo com que o algoritmo evolutivo tenha uma convergência a um resultado melhor e mais rápido.

2.3.1 Evolução dos pesos

A evolução dos pesos na rede neural evolutiva ocorre através do uso de algoritmo evolutivo, minimizando alguma função de erro específica como o erro quadrático ou entropia cruzada [31, 44], maximizar algum score específico [9].

Este tipo de Rede neural evolutiva é a mais simples, pois utiliza algoritmo evolutivo para otimizar apenas o pesos da própria rede neural. Para isto, existe algumas formas de representar os pesos para a aplicação do algoritmo genético:

- strings binárias: Os pesos são representados como um conjunto de bits com um tamanho definido por parâmetro de entrada. A vantagem desta representação é que é facilmente manipulável pelos algoritmos genéticos pois são apenas

genes binários que podem mudar entre 0 e 1. Uma das desvantagens desta representação é que o escopo de pesos que podem ser representados é limitado

- números reais: Os pesos são representados por números reais, aumentando bastante o escopo em relação as strings binárias. Porém, os operadores genéticos precisam ser alterados pois não é mais possível fazer um crossover cortando o gene pela metade.

Para os pesos com números reais, [31] definiu alguns operadores genéticos que ajudam a resolver o caso:

- Mutação *unbiased*: A mutação *unbiased* de um gene específico troca o número por outro número totalmente aleatório, sem levar em conta nenhuma informação do gene atual.

- Mutação *biased*: A mutação *biased* de um gene específico, é uma mutação supervisionada que recebe o valor atual do gene e multiplica por "fator de mutação", fazendo com o que o gene mude um pouco comparado com o resultado anterior.

- Crossover nos pesos: O *crossover* de um gene que representa o peso é feito de maneira em que é escolhido um valor entre o pai ou a mãe aleatoriamente, sem fazer nenhuma mistura entre os valores.

- Crossover dos nós: No crossover de nós, é escolhido um neurônio aleatoriamente de um dos pais e de acordo com o neurônio correspondente com o outro pai, é gerado um filho onde o neurônio escolhido é de um pai e os outros neurônios são do outro pai.

- Hill-climb: No hillclimb, é calculado o gradiente de cada membro do conjunto de treino e é feita a soma do gradiente total. Em seguida, o gradiente é normalizado fazendo uma divisão do mesmo com a sua magnetude. O filho é gerado através do pai fazendo um passo em direção determinada pelo gradiente.

2.4 Programação CUDA

A tecnologia CUDA foi uma tecnologia criada pela Nvidia [13], a qual permitia que fossem realizadas operações matemáticas dentro de sua GPU. A ideia principal da tecnologia era utilizar o alto número de núcleos que as GPUs da Nvidia possuem, com o intuito de obter um desempenho maior na execução nos algoritmos abstraindo alguns problemas que a arquitetura possui: hierarquia dos grupos de

threads, memórias compartilhadas e barreira de sincronização. Apesar destas abstrações, ao programar utilizando CUDA é necessário ter um grande conhecimento da arquitetura da GPU, pois apesar das abstrações, as limitações ainda existem.

2.4.1 Arquitetura CUDA

A arquitetura CUDA [13] é formada por dois tipos de processadores: os *Streaming Multiprocessors* (SM) e os *Scalar Processors* (SP). Em cada SM possui vários SP, o qual cada um executa apenas um *thread*, porém eles possuem uma arquitetura SIMT(*Single Instruction, multi thread*) [13], o qual todos os SP de um mesmo SM executam a mesma instrução de forma paralela. A GPU também possui mais transistores de execução do que de cache e controle de fluxo [13]. Como pode ser observado no exemplo da figura 2.15. Isto faz com que a arquitetura seja boa para resolver problemas que necessitem executar cálculos aritméticos intensivos em um grande conjunto de dados e com proporção alta de operações aritméticas por memória [13].

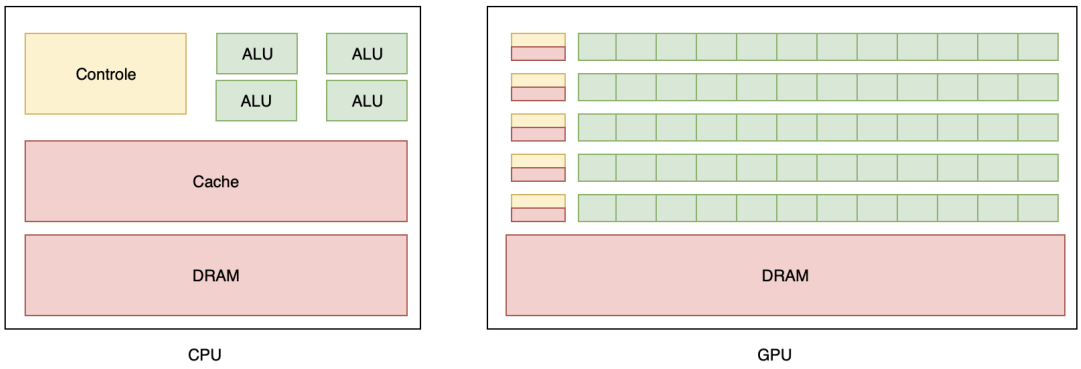

Figura 2.15: Conceito de grades e blocos em CUDA.

Para facilitar a obtenção do paralelismo nesta arquitetura, foi criado o conceito de grades e blocos.

2.4.2 Conceito de grades e blocos

Segundo a própria NVIDIA [13], o conceito de grades e blocos é utilizado para manipular os threads. Cada bloco representa um SM, e um conjunto de blocos forma uma grade. Cada bloco possui um espaço de memória próprio e é capaz de executar múltiplos threads, representadas pelos SPs. Este conceito é representado pela figura 2.16.

A ideia deste conceito é que ele escale ao paralelismo de GPUs com tamanhos diferentes. Como cada GPU possui um número de processadores diferentes, é possível

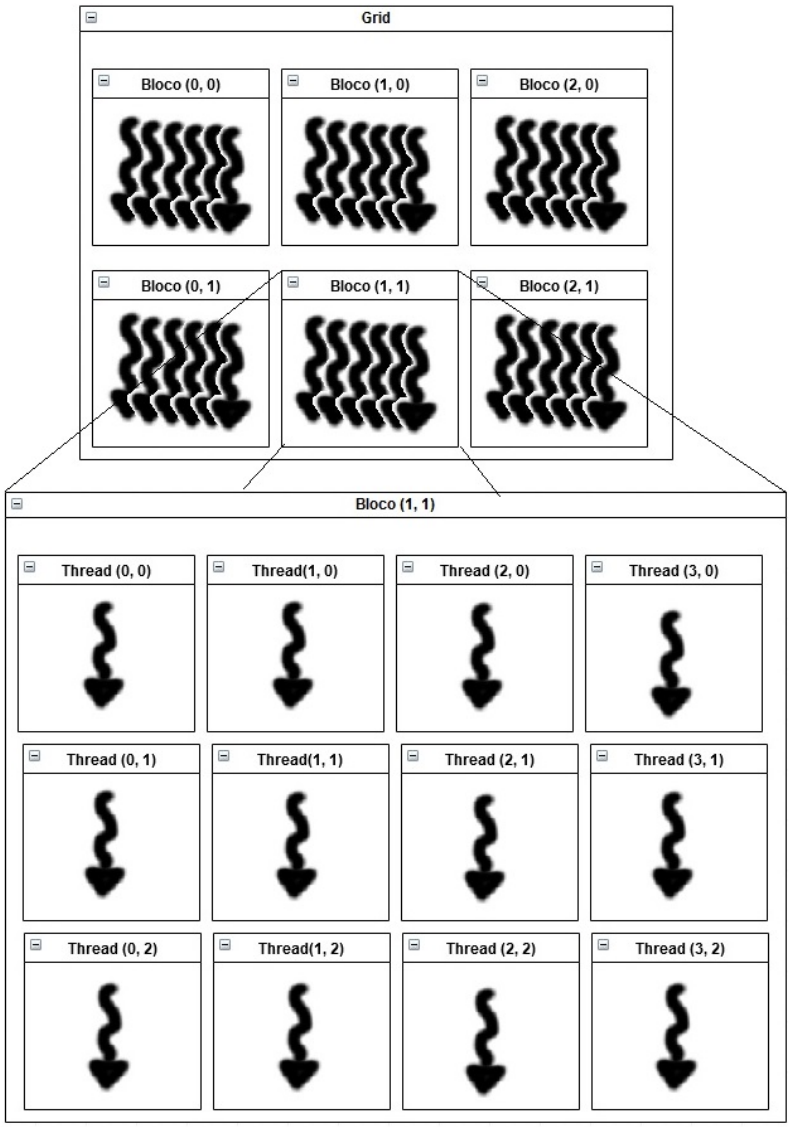

Figura 2.16: Conceito de grades e blocos em CUDA.

gerar uma aplicação que utilize o processador completamente, gerando um número de iterações que a GPU irá rodar sendo igual ao número de blocos dividido pelo número de SM que a GPU possui.

2.4.3 Memória hierárquica

Os *threads* em CUDA conseguem ter acesso de leitura e escrita a 3 tipos de espaços de memória: a memória local do *thread*, a memória local compartilhada pelo bloco onde elas estão e a memória global. Além do acesso a esses espaços de memória, elas também possui acesso apenas para leitura os seguintes: o espaço de constante e de memória de textura. A estrutura da memória hierárquica pode ser observada através da figura 2.17.

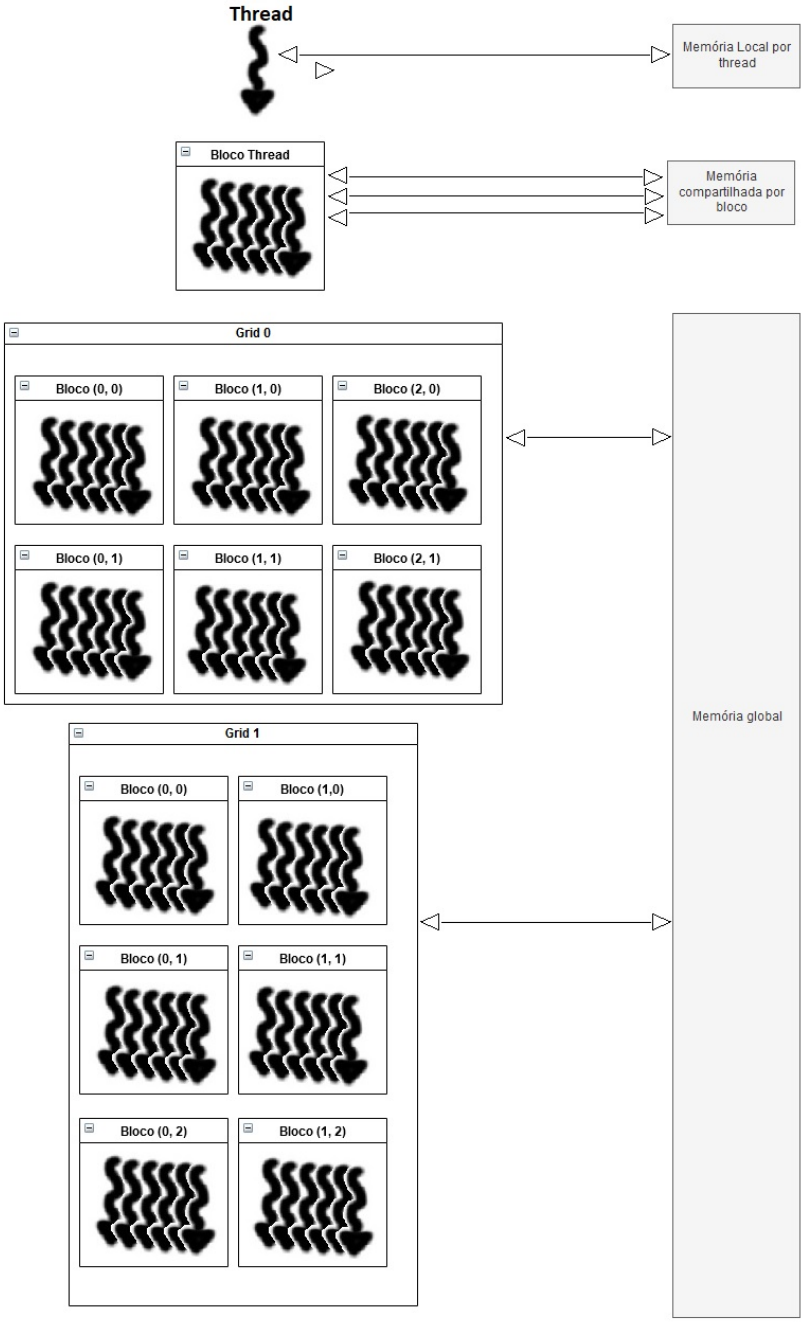

Figura 2.17: Hierarquia de memória em CUDA

Capítulo 3

Proposta

Neste capítulo será apresentado o problema de como organizar a execução da rede neural evolutiva na GPU e uma proposta do fluxo de execução que visa solucionar este problema.

3.1 Problema da Execução da rede neural evolutiva na GPU

Problemas na execução de redes neurais na GPU A execução da rede neural na GPU possui alguns desafios devido a arquitetura da própria GPU. Um deles seria a limitação de memória que a GPU impõe [45].

Outro problema seria a diferença no paradigma de programação que a GPU exige para tirar o máximo proveito da sua arquitetura [13]. Um dos principais conceitos que é necessário levar em conta é o de grades e blocos, pois deveria maximizar o número de processadores SP utilizados, os quais possuem a limitação de executar apenas a mesma instrução por SM. Outro desafio é o conceito da memória hierárquica, onde a estrutura deve ser otimizada de forma que os *threads* não precisem acessar a memória global, fazendo com que haja perda de velocidade [46].

Até a presente data, não há relato de nenhum caso sobre como utilizar a GPU para a otimização dos pesos de redes neurais através de algoritmos genéticos. Os casos encontrados utilizavam uma estrutura que beneficiava a paralelização nas CPUs, utilizando múltiplos computadores para executarem de maneira paralela [15].

3.2 Proposta de estrutura

Para atacar o problema, será proposta uma estrutura que facilita o fluxo de execução do algoritmo na GPU. O objetivo da estrutura é ajudar a vetorização das funções que são executas durante o treino do algoritmo evolutivo, principalmente no quesito

dos operadores genéticos. A estrutura proposta irá levar em conta o fato de todas as redes neurais possuírem a mesma estrutura.

Esta estrutura propõe com que cada camada da população esteja em apenas uma matriz, fazendo com que a população seja representada por n matrizes com tamanhos diferentes, onde n é o número de camadas que a rede neural irá possuir. Como cada matriz desta estrutura representará uma camada específica da rede neural de todos os indivíduos da população, esta camada será um tensor de dimensão $k + 1$ onde k é a dimensão da camada especificada que cada indivíduo irá possuir. Um exemplo pode ser visto na figura 3.1 com uma camada totalmente conectada:

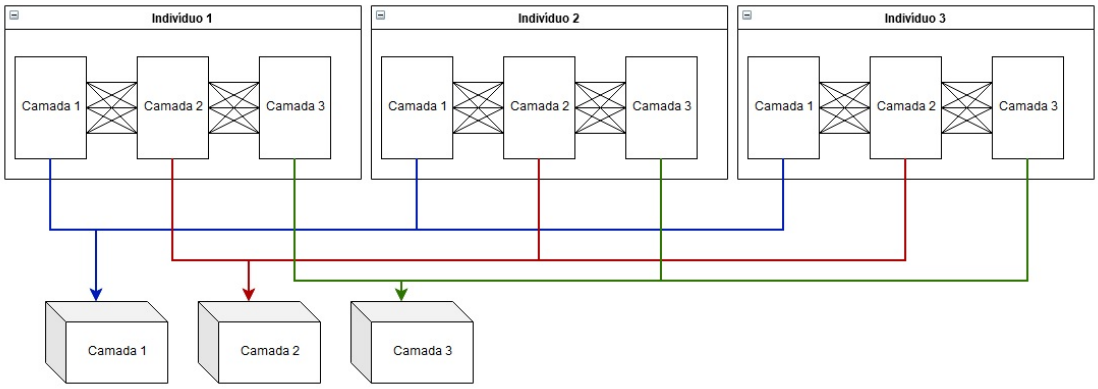

Figura 3.1: Exemplo da estrutura dos pesos de uma camada da população

Com isto, será possível calcular o fitness das redes neurais e dos operadores genéticos de forma paralela através de funções vetorizadas para uma população inteira, o que aumenta muito a velocidade de execução na GPU, pois aumenta o número de dados que podem ser processados de forma paralela na GPU.

Com a estrutura definida, iremos comparar três abordagens diferentes para calcular o fitness nas redes neurais *MLP*. Utilizaremos a função *map*, que é uma função que recebe dois parâmetros, o primeiro é um *array* de objetos e o segundo é uma função. Em seguida é executada a função de entrada para cada elemento do *array*. Na primeira abordagem é feita uma função *map* , onde irá repetir a mesma função de fitness para todas as redes neurais de forma paralela. Na segunda abordagem, iremos executar a função *map* para cada camada da rede neural, fazendo com que a execução do algoritmo seja paralelizada de camada em camada, onde o cálculo da próxima camada da rede neural será somente executado após toda a população executar a camada corrente. Esta abordagem diminui a necessidade de alocação de toda execução na placa de vídeo, limitando apenas a camada em que a população está percorrendo. Na terceira abordagem iremos utilizar a estrutura proposta para obter o máximo de desempenho possível utilizando apenas um *batch* de multiplicação de matrizes sendo aplicado em um vetor 3D, para obter o maior grau de paralelismo na GPU. As três abordagens podem ser vistas através da figura 3.2.

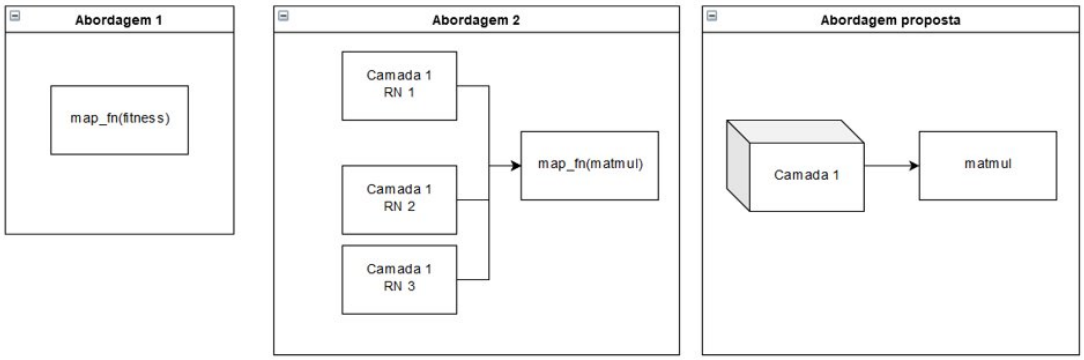

Figura 3.2: Abordagens para o cálculo do fitness nas redes neurais *feedfoward* simples

Porém, a primeira abordagem que é a mesma utilizada no trabalho de MONTANA e DAVID [31], não é possível executar na GPU, devido ao alto grau de uso de memória que a função *map* possui nesta abordagem. Isto ocorre porque a função irá alocar todas as variáveis que irão ser abordadas em algum momento dentro do *map* para cada rede neural, incluindo os resultados das execuções das camadas intermediárias. Devido a este problema, esta abordagem foi descartada do trabalho.

Para o caso das redes neurais convolucionais, teremos as mesmas três abordagens para calcular o fitness. Na primeira abordagem, iremos executar uma função *map* em cada camada da rede neural, seja ela uma camada convolucional ou uma camada totalmente conectada. Já na ultima abordagem, que será a proposta, iremos executar a função map do fitness apenas nas camadas convolucionais da rede neural, e em seguida, iremos calcular as camadas totalmente conectadas utilizando a abordagem proposta das redes neurais *MLP*, executando um *batch* de multiplicação de matrizes através de um vetor de três dimensões. As abordagens podem ser vista através da figura 3.3.

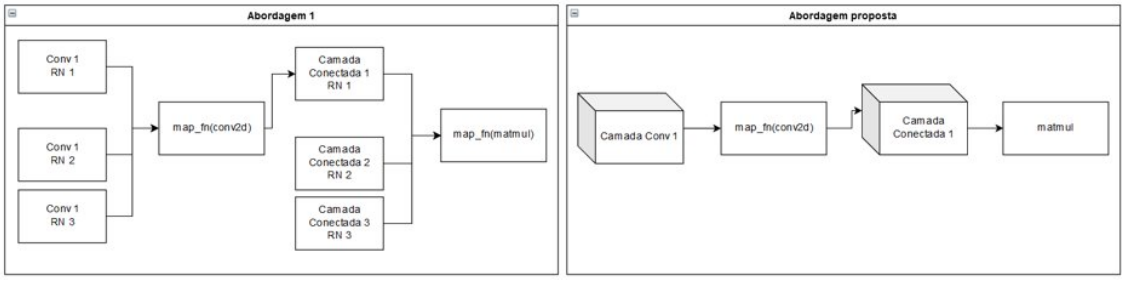

Figura 3.3: Abordagens para o cálculo do fitness nas redes neurais convolucionais

Estamos propondo o uso de duas versões para o operador de seleção:

- Seleção por Truncamento: Nesta versão não existe há muito ganho de performance em relação ao modelo paralelo, logo foi utilizado a abordagem proposta

no artigo do [31]

- Seleção por torneio: Para realizar a seleção via torneio foi a população foi dividida em N subconjuntos aleatórios com um número igual de participantes. Dentro de cada subconjunto, é selecionado apenas o indivíduo que possua o maior *fitness* entre os membros do mesmo.

Também serão propostos alguns operadores genéticos que irão fazer o uso desta estrutura de forma vetorizada, demonstrando que é possível obter grandes ganhos de velocidade com as operações corretas. Serão propostos 4 operadores baseados no trabalho de MONTANA e DAVID [31] para serem aplicados nos pesos das redes neurais:

- Mutação *unbiased* : Para a mutação *unbiased*, iremos criar um tensor aleatório do mesmo tamanho de cada camada o qual representará uma máscara que definirá os valores que serão modificados pela mutação. Em seguida será gerado outro tensor aleatório do mesmo tamanho de cada camada que irá definir qual será o novo valor que o gene mutado irá receber entre 0 e 1. Com isto, é só aplicar os valores do tensor no tensor que irá ser mutado onde o valor da máscara é positivo.

- Mutação *biased*: Para a mutação *biased*, o algoritmo irá se comportar da mesma maneira que a mutação *unbiased*. Só que desta vez, os tensores aleatórios que irão definir os novos valores serão tensores com valores entre 1 - x e 1 + x, onde x é o parâmetro de entrada do operador. Em seguida, estes tensores serão multiplicados pelos valores dos pesos originais da rede neural.

- Crossover com máscara: Iremos gerar uma uma máscara que será representada por N tensores com valores aleatórios entre 0 e 1 onde cada tensor irá possuir o tamanho de cada camada da rede neural. Em seguida, o valor do próximo tensor que irá receber será o valor da mãe multiplicado pelo valor aleatório somado com o valor do pai multiplicado por $1-$ valor aleatório. Com isto iremos gerar um filho que irá possuir os valores mistos do pai e da mãe.

Com estas operações, é possível alterar o fenótipo das redes neurais de forma direta, sem precisar converter os genes para as redes neurais. Logo, ao alterar o peso de uma rede neural, estará alterando diretamente o gene da mesma, fazendo com que diminua a complexidade da transformação dos genes para redes neurais na placa de vídeo. Também, utilizando a estrutura proposta para as redes neurais, é possível vetorizar os operadores genéticos de forma seja possível aplicá-los em toda população de forma paralela e obter um desempenho maior na GPU. Um exemplo disto é executar a mutação no tensor inteiro da camada que representa

uma população de 60 indivíduos em vez de percorrer cada individuo e realizar a mutação, como pode ser visto nos pseudocódigos representados pelos algoritmos 1 e 2 em uma rede neural *MLP* com 3 camadas.

Algorithm 1: Mutação com os tensores agrupados

```
inputs : população representada pela estrutura proposta
foreach camada do
    camada ← mutação
end
```

Algorithm 2: Mutação com os tensores desagrupados

```
inputs : um array da população em que cada individuo possui um array que
         onde cada elemento representa uma camada da rede neural
foreach indivíduo do
    foreach camada do
        camada ← mutação
    end
end
```

Capítulo 4

Avaliação Experimental

Neste capítulo foram realizados vários experimentos em um *dataset* com o intuito de verificar se a estrutura proposta no trabalho beneficia a velocidade de execução da otimização das redes neurais *MLP* e convolucionais utilizando algoritmo genético. Em um *dataset* utilizado para classificação, treinamos duas redes neurais diferentes variando a estrutura, onde uma é a rede neural *MLP* e outra é a rede neural convolucional, a fim de comparar o tempo de treinamento de ambas utilizando algoritmo genético para otimizar os pesos.

Este capítulo possui a seguinte estrutura: Primeiro será discutido quais são os objetivos e como serão executados os experimentos. Em seguida, a metodologia será discutida para a avaliar as propostas do experimento, descrevendo os *datasets* que foram utilizados. Também será mostrado qual a configuração utilizada nos experimentos e quais métricas serão avaliadas. Em seguida apresentaremos qual a tecnologia utilizada, onde será descrito quais ferramentas utilizadas e o ambiente em que os testes foram executados. Por fim será demonstrado os resultados e suas respectivas análises.

4.1 Objetivo dos experimentos

O objetivo dos experimentos é aferir o ganho de velocidade com a utilização da estrutura em relação a execução do treino utilizando o algoritmo genético com a estrutura definida em [31]. Para isto, serão avaliados problemas de classificação de imagens, onde serão executados vários cenários para comparar o tempo de execução com o desempenho. Os testes serão executados em um ambiente igual para que seja possível avaliar o ganho apenas da utilização da estrutura. Dado isto, serão realizados os seguintes experimentos:

- Experimento 1: Execução do treino das redes neurais *MLP* e convolucionais seguindo a estrutura de organização das redes neurais de acordo com [31]

- Experimento 2: Execução do treino das redes neurais *MLP* e convolucionais utilizando a estrutura proposta no trabalho.

Também serão avaliados o desempenho separado de cada parte da execução da rede neural evolutiva em três partes: fitness, seleção e operações genéticas. O objetivo desta divisão é poder verificar o ganho e a perda obtida em cada uma das etapas da execução do algoritmo.

4.2 Metodologia

Nesta seção apresentaremos a metodologia utilizada para avaliar a proposta do trabalho. Serão abordados os seguintes tópicos: O *dataset* utilizado; a configuração do experimento com seus parâmetros e técnicas aplicadas no algoritmo; e por último, as métricas utilizadas.

4.2.1 Dataset

MNIST

Será utilizado o *dataset* MNIST [47] , sendo este um *dataset* que possui números escritos a mão. Ele é uma versão menor do *dataset* NIST para o reconhecimento de texto escrito a mão. Este *dataset* é composto por imagens 24x24 em que cada uma possui um número escrito de zero a nove. O conjunto de treino possui 60,000 imagens e o conjunto de testes possui 10,000. O objetivo deste *dataset* é fazer com que o algoritmo seja capaz de identificar qual número que está sendo representado em uma imagem. As imagens foram tratadas de forma que assumissem uma representação binária, de maneira que adquirisse o valor "1"onde o tom de preto fosse maior, e "0"onde o tom de branco fosse maior.

4.2.2 Configuração dos experimentos

O conjunto de treino do *dataset* foi dividido em duas partes de forma aleatória: 90% para treino e 10% para validação. Com esta divisão, o *dataset* MNIST ficará com 54,000 imagens para treino, 6,000 para validação e 10,0000 para teste. O conjunto de validação será utilizado para verificar como o erro está se comportando durante a execução do treinamento com o intuito de verificar o comportamento do algoritmo sem ter que utilizar o conjunto de teste fazendo com que o nosso experimento não represente o resultado real.

Utilizamos 4 estruturas para as redes neurais, que possuíam a função de ativação sigmoide ou tangente hiperbólica de acordo com o experimento:

- Rede neural *MLP*com 3 camadas ocultas.
- Rede neural
  textitMLP com 6 camadas ocultas.
- Rede neural convolucional com 3 camadas, sendo a primeira uma convolução, a segunda uma de agregação e a última uma camada totalmente conectada.
- Rede neural convolucional com 6 camadas

Os pesos das redes neurais foram inicializados utilizando o inicializador de HE normal [48], conforme foi utilizado em vários trabalhos [49–51]. Neste inicializador, o valores da primeira camada são gerados aleatoriamente seguindo uma distribuição normal. Em seguida, os valores das próximas camadas são gerados também com uma distribuição normal, porém esta distribuição possui um desvio padrão de fórmula abaixo, onde *fanIn* é o número de conexões que ela recebe da camada anterior.

$$\text{STDEV} = \sqrt{\frac{2}{fanIn}}$$

Iremos comparar o desempenho do algoritmo após a execução de 800 gerações da mesma forma executada por Montana em [31]. O *dataset* deverá ser dividido em *batchs* com tamanho de 128 entradas para que o algoritmo possa ser executado na GPU.

O algoritmo genético irá receber as seguintes características:

- Codificação do cromossoma: os pesos e os *biases* serão codificados como uma lista de valores reais. Para a estrutura proposta, os valores serão trabalhados diretamente na estrutura para o individuo que ela representa.
- Função *Fitness*: A função fitness que irá avaliar os indivíduos será a entropia cruzada da rede neural gerada (como pode ser visto na sub-seção seguinte).
- Procedimento de inicialização: Os pesos das redes neurais serão iniciados utilizando o inicializador de He Normal [48], o qual obteve bons resultados para o reconhecimento de imagens.
- Operadores: Foram utilizados os seguintes operadores em nossos experimentos:
  - Crossover com máscara: Neste operador o filho ira ter os pesos dos pais através de uma máscara. Esta máscara possui um valor entre 0 e 1 definindo a porcentagem do peso que o filho irá receber dos pais.

  - Mutação *Unbiased*: A mutação é feita utilizando uma máscara binária inicial o qual irá definir quais genes serão alterados. Em seguida dentro desta máscara são colocados valores aleatórios onde a máscara possuiu valor 1. Em seguida é aplicada a máscara na matriz de pesos substituindo todos os pesos onde a máscara possuiu valor diferente de zero.
  - Mutação *Biased*: Esta mutação é realizada criando uma máscara inicialmente com valores enter -1 e 1. Esta máscara é aplicada a matriz de peso, multiplicando o peso atual pelo valor da máscara.

- Número de indivíduos que serão gerados a partir da utilização de cada tipo de operador: O algoritmo irá possuir uma lista com o números de indivíduos que o operador irá gerar, onde cada elemento desta lista representa o número de indivíduos para os quais o algoritmo irá executar com um operador específico para gerar novos indivíduos na população. De acordo com a execução de cada iteração do algoritmo genético, os indivíduos gerados pela execução com os operadores serão avaliadas conforme o melhor resultado da função *fitness*. Considerando x como a taxa de *fine-tuning*, a contagem de execução de cada tipo de será incrementada em x para cada melhor resultado obtido durante as operações, e decrementada de x para cada pior resultado. Caso o operador na lista possua uma contagem de execução menor ou igual a x, o algoritmo irá diminuir em x da contagem do operador da lista que teve o segundo pior resultado, garantindo que o algoritmo execute com cada operador em pelo menos 1 individuo na população. Este algoritmo é parecido com a abordagem de [31], porém, o algoritmo irá executar cada operador genético baseado na lista que define a quantidade de vezes que o operador deve ser executado. Nos experimentos deste trabalho,3/6 a taxa de fine-tuning será 1.

- Tamanho da população: Variamos o tamanho da população em 20, 40 e 60, com o intuito de verificar o impacto do desempenho do algoritmo conforme aumenta o tamanho da população.

Como o *dataset* MNIST possui imagens de 28x28, então as redes neurais irão possuir 784 *features* de entrada. Com isto, a camada inicial para este *dataset* irá possuir 784 neurônios, onde cada neurônio irá receber a *feature* de cada pixel. A quantidade de camadas ocultas também irá variar de acordo com os experimentos e cada camada densa irá possuir 800 neurônios nas redes neurais *MLP*, conforme apontado em [52], e nas camadas convolucionais filtros de tamanho 3x3 com 36 canais de entrada e saida, para que consiga manter uma camada densa de 800 neurônios para comparar com as redes *MLP*. A camada de saída irá possuir 10 neurônios, onde cada um destes neurônios irá representar uma das classes de saída, ou seja, o dígito

de 0 a 9. Nesta ultima camada de ativação será aplicado a função de *softmax*, o qual normaliza todas as n saídas para uma para uma distribuição com n probabilidades, onde todas as probabilidades possuem o valor maior que zero e a soma de todas estas probabilidades vale 1 [53].

4.2.3 Métricas

Para a avaliação dos experimentos nos *datasets* de classificação, como no caso do MNIST, será utilizado a Perda de Entropia Cruzada (*Cross-Entropy*), o qual é utilizada para averiguar o desempenho das redes neurais em problemas de classificação [54]. Esta métrica calcula a diferença entre duas distribuições com probabilidades p e q. A fórmula para calcular esta métrica pode ser descrita como:

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)$$

Esta métrica possui uma vantagem em relação ao erro quadrático em problemas de classificação, pois ajuda a achar pontos ótimos melhores do que o erro quadrático [55]. A entropia cruzada é utilizada em vários trabalhos que utilizam redes neurais como em [52], [56] e [57].

Também será utilizada a métrica de média do erro quadrático médio (*Mean Squared Error*), utilizado em [31]. Esta métrica calcula a média da diferença entre o resultado obtido pelo algoritmo de *machine learning* em comparação com o resultado real elevado ao quadrado. A fórmula pode ser vista descrita em:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Também foi utilizado para critério de comparação no *dataset* MNIST a acurácia, a qual é definida pela fórmula:

$$\text{Acurácia} = \frac{\#\{\text{Número de classes acertadas}\}}{\#\{\text{Tamanho do conjunto de entrada}\}}$$

4.2.4 Tecnologia Utilizada

Todos os experimentos utilizaram a linguagem de programação python 3.7 utilizando as bibliotecas Tensorflow-GPU, Numpy, time, matplotlib e pickle. A execução do workflow na GPU foi feita utilizando a biblioteca Tensorflow-GPU 1.13 [58], o qual utiliza o framework CUDA [13] para a execução das suas operaçõesa . A geração dos

gráficos foi feita utilizando a biblioteca Matplolib [59]. O ambiente computacional utilizado foi um computador equipado com processador *Intel Core* I7-7740X com *clock* de 4.30GHz, 32GB de memória *RAM* e uma placa de vídeo Geforce 1080 Ti. A *CPU* possui 4 núcleos e é capaz de executar 8 threads utilizando a tecnologia *hyperthread*. O *GPU* desta máquina possui a arquitetura *Pascal* com 3584 *cores* CUDA.

4.3 Resultados e análise

Nesta seção será demonstrado os resultados obtidos e análises a partir dos experimentos executados utilizando as configurações demonstradas anteriormente.

4.3.1 Experimento 1

Neste experimento, foi utilizada uma rede neural evolutiva utilizando a abordagem definida pelo [31], com a diferença que utilizamos o *dataset* MNIST com as redes neurais definidas com as configurações definidas na subseção anterior. Neste experimento iremos verificar o comportamento do algoritmo genético variando o tamanho da população e o tamanho do *batch*, também comparando a utilização da função sigmoide contra a tangente hiperbólica em um espaço de execução de apenas 800 épocas, mesmo número de épocas executadas no experimento de [31]. Iremos utilizar um *batch* com tamanho de 128 entradas e a utilização de entropia cruzada como fitness.

Dez gerações do algoritmo foram executadas utilizando apenas a CPU para verificar posteriormente qual foi o *speedup* obtido nas abordagens que utilizaram a GPU. Os resultados desta execução para todas as redes que irão ser abordadas neste experimento pode ser verificado na tabela 4.1.

O comportamento obtido durante o treino do experimento pode ser verificado na figura 4.1. Como podemos observar, a utilização de 3 camadas ocultas gerou custo bem menor que a utilização de apenas 6 camadas ocultas dentro das 800 gerações executadas. Conforme visto no comportamento, ocorreram algumas oscilações do valor da função de custo durante o treino. Também podemos verificar que o algoritmo demorou para começar a convergir, podendo demorar até 80% do tempo total de execução até o custo começar a decair. Também podemos verificar que as redes neurais de 6 camadas não conseguiram convergir de uma maneira boa em relação as redes neurais com 3 camadas, provavelmente devido a estrutura da rede neural não ser ideal para este problema utilizando o algoritmo genético como otimizador

Para as redes convolucionais, obtemos o comportamento demonstrado na figura 4.2 utilizando um *batch* com tamanho de 128 entradas. Diferentemente das redes

Tamanho da População	Rede Neural	Tempo (em segundos)
20	*MLP* com 3 camadas	602,91
	MLP com 6 camadas	988,26
	CNN com 3 camadas	1841,34
	CNN com 6 camadas	1890,05
40	*MLP* com 3 camadas	1311,01
	MLP com 6 camadas	2172,93
	CNN com 3 camadas	3864,07
	CNN com 6 camadas	3737,07
60	*MLP* com 3 camadas	1961,77
	MLP com 6 camadas	3250,28
	CNN com 3 camadas	5801,01
	CNN com 6 camadas	5448,30

Tabela 4.1: Tabela com os desempenhos obtidos utilizando apenas a CPU para apenas dez gerações

neurais *MLP* o algoritmo começou a convergir bem mais rápido. O tamanho da população afetou diretamente o desempenho do algoritmo genético, neste caso as populações maiores conseguiram obter um resultado melhor que as populações menores, onde as redes neurais convolucionais de 3 e 6 camadas possuem um resultado bem próximo usando uma população de 60 individuos. É possível notar que houve mais oscilações utilizando esta estrutura do que a rede neural *MLP*. Outra coisa importante notar é que o tempo de treino de uma rede neural convolucional é menor que uma *MLP*, provavelmente pelo tamanho do cromossomo também ser menor, fazendo com que as operações genéticas sejam mais rápidas.

Por fim, o desempenho final do algoritmo pode ser observado pela tabela 4.2. Como pode ser obersavado, as redes neurais convolucionais obtiveram um desempenho bem melhor que as redes neurais *MLP*, assim como tiveram um tempo de treinamento menor no mesmo número de épocas. Também podemos notar que o uso da entropia cruzada como função de custo refletiu diretamente na acurácia do algoritmo, onde o valor resultado pelas funções de custo refletiram diretamente em uma acurácia maior, isto pode ser comprovado em todos os casos demonstrados. As redes neurais *MLP* com 3 camadas obtiveram um resultado melhor do que as redes *MLP* com 6 camadas em todos os tamanhos de populações, Porém, isto não foi visto nas redes neurais convolucionais, onde na população de tamanho 60 a rede convolucional de 3 camadas superou a rede convolucional de 6 camadas.

Em critérios de velocidade, considerando que o tempo de execução das gerações são iguais, podemos multiplicar o valor de uma época da CPU por 100 e ter uma estimativa de quanto tempo demoraria executar 100 gerações na CPU. Com isso, podemos verificar que esta abordagem utilizando a GPU teve um grande *speedup*. A comparação destes resultados pode ser visto em 4.3. Podemos perceber que houve

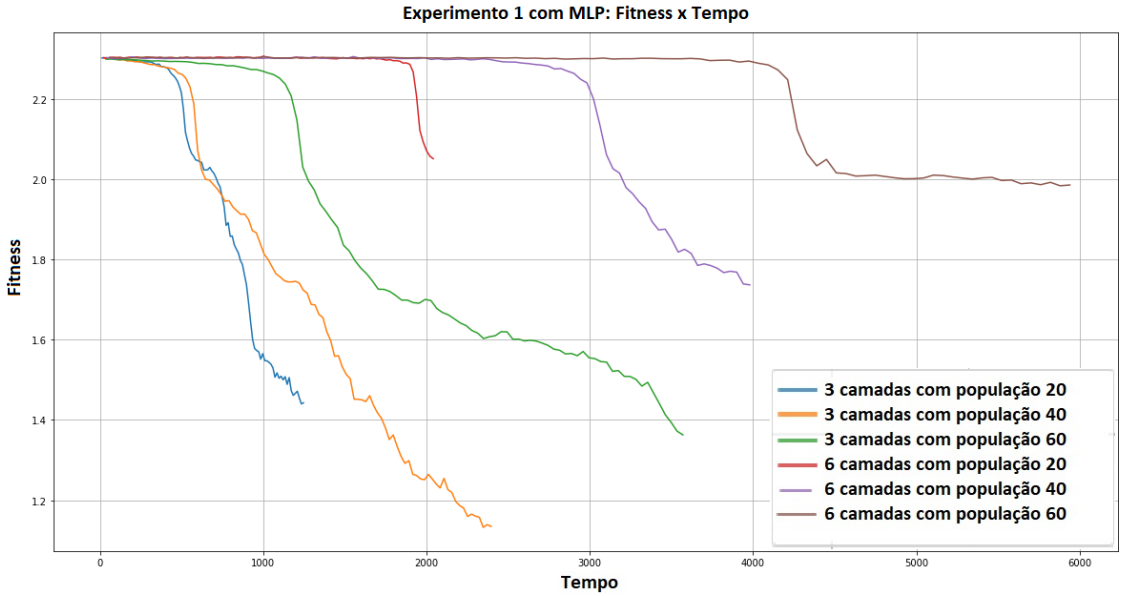

Figura 4.1: Melhor *fitness* das redes neurais *MLP* com função sigmoide

um ganho de desempenho considerável em todos os casos, principalmente nas redes neurais convolucionais, onde a taxa de *speedup* alcançou acima de 10 em todos os casos. Porém, o ganho não foi o mesmo nas redes neurais *MLP*, fazendo com que a taxa máxima de *speedup* fosse apenas 5,49.

Para uma análise melhor do algoritmo, foi obtido o tempo de execução de cada parte do algoritmo genético, como pode ser visto na tabela 4.4. Como podemos notar, as etapas que mais consomem o tempo de execução são respectivamente as operações genéticas e cálculo do fitness. É possível notar que o aumento do tamanho da população influencia quase linearmente no tamanho do cálculo do fitness, como visto nas redes *MLP* e convolucionais, porém, não pode ser dito o mesmo para as operações genéticas.

Para fazer uma análise melhor de como o algoritmo foi executado na GPU, foi retirada a linha de tempo de execução do *workflow* na GPU. O resultado pode ser visto na figura A.1 que está no anexo. A figura 4.3 representa parte da execução do algoritmo na rede neural *MLP* com 3 camadas com a população de 40 indivíduos e as figuras 4.4 e 4.5 representam a parte da execução no fitness na camada convolucional e na totalmente conectada em uma rede neural convolucional de 3 camadas também com uma população de 40 indivíduos. Como podemos observar no caso da rede neural *MLP*, devido à falta de vetorização nas multiplicações de matrizes executadas nas camadas das redes no cálculo do *fitness*, a execução do algoritmo na GPU sofreu perdas de performance pois era necessário iterar sobre população em cada camada para executar a multiplicação de matrizes, impactando o uso total da GPU durante o cálculo do fitness. Isto pode ser observado principalmente nos espaços criados após a cada execução da multiplicação de matrizes, representadas pelos espaços

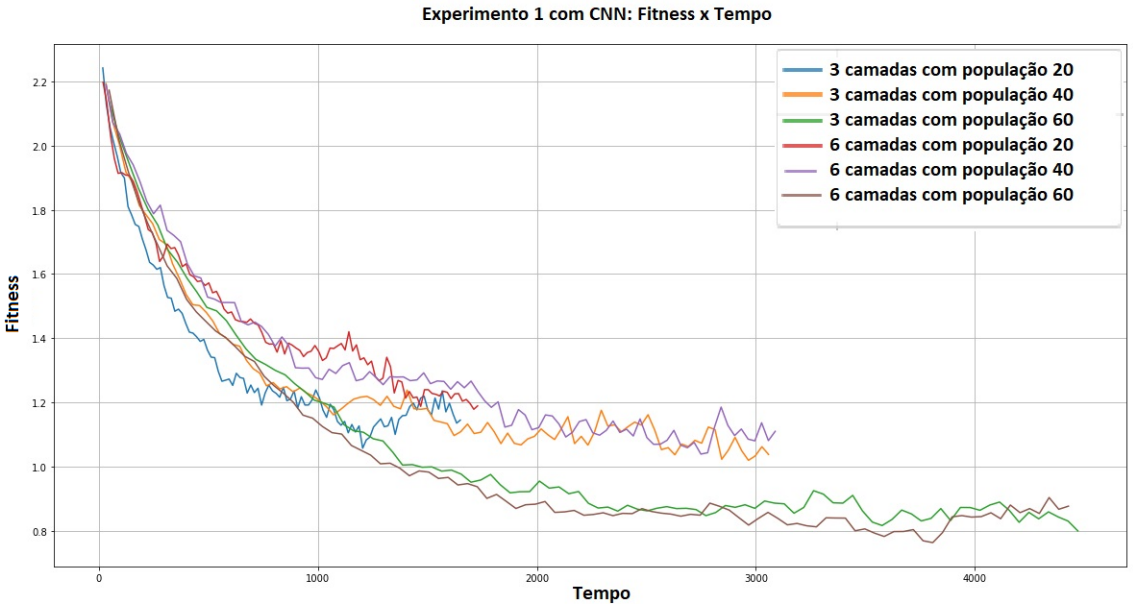

Figura 4.2: Melhor *fitness* das redes neurais convolucionais com função sigmoide

em vermelho e a soma representadas pelos espaços em amarelo. Também podemos observar que o caso se repete na rede convolucional porém em um grau bem menor devido a dimensão dos pesos convolucionais serem menores também, mas ainda podemos observar os espaços criados tanto nas camadas convolucionais como nas camadas totalmente conectadas. Dado estes resultados, vemos uma necessidade de uma estruturação melhor das redes neurais e do problema a fim de que seja possível fazer um uso melhor da GPU eliminando estes espaços onde a placa de vídeo não estava executando nenhuma operação.

4.3.2 Experimento 2

Neste experimento, será utilizada a estrutura proposta neste trabalho, onde todos os pesos de uma determinada camada na população serão concatenados para virar um tensor único. Nele iremos analisar os mesmos dados analisados no experimento anterior e iremos comparar se houve uma melhoria de desempenho, utilizando o mesmo critério para as redes neurais que serão executadas em apenas 100 épocas.

Seguindo como abordado no experimento anterior, o comportamento obtido pelas redes neurais *MLP* com *batch* de tamanho 128 pode ser visto na figura 4.6 e para as redes neurais convolucionais utilizando o mesmo tamanho de batch na figura 4.7 . Como podemos observar, o comportamento obtido foi o mesmo do experimento 1 mas em um tempo de execução menor, demonstrando que o uso da estrutura ajudou na execução do treino sem afetar o funcionamento do algoritmo genético.

Na tabela 4.5 , podemos notar que houve uma melhoria expressiva no cálculo do fitness para as redes neurais *MLP*. Podemos concluir que o uso da estrutura

Tamanho da População	Rede neural	Custo	Acurácia	Tempo (em segundos)
20	*MLP* com 3 camadas	1,442	0,492	1245,952
	MLP com 6 camadas	2,051	0,188	2040,809
	CNN com 3 camadas	1,145	0,644	1650,233
	CNN com 6 camadas	1,189	0,602	1729,142
40	*MLP* com 3 camadas	1,1347	0,63	2395,321
	MLP com 6 camadas	1,736	0,375	3979,75
	CNN com 3 camadas	1,038	0,654	3057,807
	CNN com 6 camadas	1,110	0,656	3087,955
60	*MLP* com 3 camadas	1,362	0,544	3569,892
	MLP com 6 camadas	1,985	0,272	5939,73
	CNN com 3 camadas	0,800	0,768	4469,986
	CNN com 6 camadas	0,877	0,730	4427,002

Tabela 4.2: Tabela com os desempenhos obtidos com *batch* 128

Tamanho da População	Rede Neural	Tempo	Tempo Teórico da cpu em 100 gerações	Speedup (CPU / GPU)
20	*MLP* com 3 camadas	1245,952	6029	4,83
	MLP com 6 camadas	2040,809	9883	4,84
	CNN com 3 camadas	1605,233	18413	11,47
	CNN com 6 camadas	1729,142	18901	10,93
40	*MLP* com 3 camadas	2395,321	13110	5,47
	MLP com 6 camadas	3979,75	21729	5,46
	CNN com 3 camadas	3057,807	38641	12,63
	CNN com 6 camadas	3087,955	37371	12,10
60	*MLP* com 3 camadas	3569,892	19618	5,49
	MLP com 6 camadas	5939,73	32503	5,47
	CNN com 3 camadas	4469,986	58010	12,97
	CNN com 6 camadas	4427,002	54483	12,30

Tabela 4.3: Tabela de comparação do tempo da CPU com a GPU

melhorou bastante no cálculo do fitness nas redes neurais *MLP*, onde o *speedup* conseguiu ultrapassar 22. Porém, este ganho não refletiu de maneira igual nas redes neurais convolucionais, fazendo com que o ganho de performance tenha sido bem menor. Houve também uma diminuição no tempo de execução das operações genéticas, devido a facilidade de execução por estar dentro do mesmo tensor.

Comparando o tempo de execução com a implementação em CPU do mesmo caso do experimento 1. Podemos ver na tabela 4.6 que a taxa de *speedup* aumentou em relação ao experimento 1, onde speedup mínimo atual é de aproximadamente 6,29 nas redes *MLP* e de 13,22 nas redes convolucionais. No experimento anterior é de 4,83 para as redes *MLP* e 10,93 para as redes neurais convolucionais. Isto representa cerca de 30% de melhoria para o caso das redes neurais *MLP* em relação

Tamanho da população	Rede neural	Fitness	Seleção	Operações Genéticas
20	*MLP* 3 camadas	43 ms	493 us	69,2 ms
	MLP 6 camadas	68,6 ms	666 us	110 ms
	CNN com 3 camadas	77,6 ms	545 us	108 ms
	CNN com 6 camadas	88,7 ms	747 us	134 ms
40	*MLP* 3 camadas	83,8 ms	552 us	111 ms
	MLP 6 camadas	148 ms	757 us	193 ms
	CNN com 3 camadas	156 ms	536 us	182 ms
	CNN com 6 camadas	198 ms	667 us	241 ms
60	*MLP* 3 camadas	133 ms	527 us	160 ms
	*MLP*6 camadas	235 ms	770 us	276 ms
	CNN com 3 camadas	231 ms	592 us	259 ms
	CNN com 6 camadas	276 ms	876 us	325 ms

Tabela 4.4: Tabela com o tempo gasto em cada etapa do algoritmo genético

Tamanho da população	Rede neural	Fitness	Seleção	Operações Genéticas
20	*MLP* 3 camadas	1,96 ms	635 us	25,8 ms
	MLP 6 camadas	68,6 ms	766 us	43,2 ms
	CNN com 3 camadas	77,6 ms	538 us	83,5 ms
	CNN com 6 camadas	88,7 ms	865 us	118 ms
40	*MLP* 3 camadas	83,8 ms	562 us	25,5 ms
	MLP 6 camadas	148 ms	790 us	38,4 ms
	CNN com 3 camadas	156 ms	603 us	127 ms
	CNN com 6 camadas	198 ms	852 us	185 ms
60	*MLP* 3 camadas	963 ms	527 us	28,7 ms
	MLP 6 camadas	1,48 ms	852 us	50,5 ms
	CNN com 3 camadas	231 ms	699 us	182 ms
	CNN com 6 camadas	276 ms	930 us	262 ms

Tabela 4.5: Tabela com o tempo gasto em cada etapa do algoritmo genético utilizando a estrutura proposta

ao experimento anterior.

A *timeline* de execução da GPU também pode ser visto de acordo com a figura 4.8, o qual é possível verificar uma utilização maior da GPU na execução do algoritmo em comparação com o experimento anterior. Podemos notar que os "espaços"gerados pela a utilização da função *map* sumiram, fazendo com que a placa de vídeo tenha uma utilização maior e diminuindo o tempo de execução. Com isto é possível reforçar que a estrutura beneficiou o desempenho do algoritmo. Já no caso das redes neurais convolucionais, podemos notar nas figuras 4.9 e **??**, que reduziu os espaços entre a execução das convoluções e das camadas totalmente conectadas. Com isto, podemos concluir que a arquitetura beneficiou a execução de todas as camadas da rede neural convolucional.

Tamanho da População	Rede Neural	Tempo	Tempo Teórico da cpu em 100 gerações	Speedup (CPU / GPU)
20	*MLP* com 3 camadas	958,267	6029	6,29
	MLP com 6 camadas	1526,7	9883	6,47
	CNN com 3 camadas	1389,03	18413	13,25
	CNN com 6 camadas	1428,98	18901	13,22
40	*MLP* com 3 camadas	1754,14	13110	7,47
	MLP com 6 camadas	2828,28	21729	7,68
	CNN com 3 camadas	2657,72	38641	14,53
	CNN com 6 camadas	2669,17	37371	14,00
60	*MLP* com 3 camadas	2566,67	19618	7,64
	MLP com 6 camadas	4110,2	32503	7,90
	CNN com 3 camadas	3950,46	58010	14,68
	CNN com 6 camadas	3877,51	54483	14,05

Tabela 4.6: Tabela de comparação do tempo da CPU com a GPU utilizando a estrutura proposta

4.3.3 experimento 3

Neste experimento iremos verificar o comportamento do otimizador genético utilizando um *batch* de tamanho 128 , utilizando o erro quadrático médio como fitness e utilizando a seleção por truncamento. Utilizando a solução proposta, o comportamento obtido pode ser visto na figura 4.11 para as redes *MLP*. Para este caso, o erro quadrático não ajudou as redes neurais convergirem. Também podemos afirmar que apenas um caso conseguiu um bom avanço que foi observado nas redes neurais de 6 camadas utilizando uma populações de 40 e 60. Outro problema notado é que o resultado teve mais oscilações que a utilização da entropia cruzada como função de fitness.

Nos casos das redes neurais convolucionais, o comportamento pode ser visto de acordo com a figura 4.12. Como podemos notar no gráfico, as redes com populações maiores conseguiram convergir melhor do que as redes menores, porém ainda continuou um grande número de oscilações nos custos.

Os resultados finais podem ser vistos na tabela 4.7. Nela observarmos que o SME refletiu diretamente na acurácia do algoritmo, porém, como o algoritmo não conseguiu convergir satifastoriamente, todas as acurácias apresentaram resultados baixos. É possível concluir que utilizando as configurações do experimento, a utilização do SME como função de fitness não foi eficaz para treinar as redes neurais no dataset MNIST.

Analisando o comportamento do algoritmo na GPU através da figura 4.13 para as redes neurais *MLP*, é possível concluir que o comportamento do algoritmo no quesito desempenho foi o mesmo do experimento 2. Com isto, é possível concluir

Tamanho da População	Rede Neural	Fitness	Acurácia	Tempo
20	*MLP* com 3 camadas	8,61233	0,0945	2.097,69
	MLP com 6 camadas	8,0985	0,0785	4.153,90
	CNN com 3 camadas	8,5475	0,0926667	6.184,18
	CNN com 6 camadas	8,56367	0,0863333	1.324,29
40	*MLP* com 3 camadas	6,51883	0,0516667	2.568,09
	MLP com 6 camadas	6,88933	0,056	3.831,44
	CNN com 3 camadas	7,59333	0,1575	1.662,71
	CNN com 6 camadas	5,88533	0,134	3.156,73
60	*MLP*com 3 camadas	4,843	0,1445	4.746,16
	MLP com 6 camadas	7,47917	0,0836667	1.746,62
	CNN com 3 camadas	5,60217	0,146833	3.207,10
	CNN com 6 camadas	4,33733	0,2205	4.620,63

Tabela 4.7: Tabela com os desempenhos obtidos utilizando SME sem utilizar a estrutura proposta

que a mudança do fitness de entropia cruzada para *SME* não influenciou em como o algoritmo foi executado na GPU. Isto também pode ser verificado nas redes convolucionais através das figuras 4.15 e **??**, onde o comportamento foi o mesmo do experimento 2 para as redes neurais convolucionais.

Rede Neural	Perda	Acurácia	Tempo
MLP com 3 camadas	0,196924	0,9476	538,637
MLP com 6 camadas	0,599673	0,8955	758,099
CNN com 3 camadas	0,843478	0,9711	2.052,27
CNN com 6 camadas	0,120511	0,9658	1.710,6

Tabela 4.8: Tabela com os desempenhos obtidos utilizando SME sem utilizar a estrutura proposta

4.3.4 Experimento 4

Neste experimento verificamos o desempenho do otimizador ADAM no *dataset* MNIST utilizando um *batch* com tamanho de 128,uma taxa de aprendizado de 0,00001 no otimizador dentro de 800 épocas, igualmente executados nos experimentos anteriores e utilizando a entropia cruzada como função de custo.

O comportamento do treino pode ser visto na figura 4.16 para as redes neurais *MLP* simples e 4.17 para as redes neurais convolucionais. As redes neurais convolucionais conseguiram fazer o custo a cair mais rapidamente do que as redes neurais *MLP*. Também podemos notar que o número menor de camadas ajudou o custo a cair mais rápido que um número de camadas maior. A queda brusca no custo das redes neurais *MLP* simples ocorreu bem mais rápido que a utilização do algoritmo genético, mesmo utilizando as mesmas estruturas.

Também podemos observar os resultados finais obtidos na tabela 4.8. Os resultados obtidos utilizando o otimizador ADAM foram melhores do que a utilização do otimizador evolutivo utilizado nos experimentos anteriores.

Figura 4.3: Comportamento do cálculo do *fitness* da redes neural *MLP* de 3 camadas com população de 40

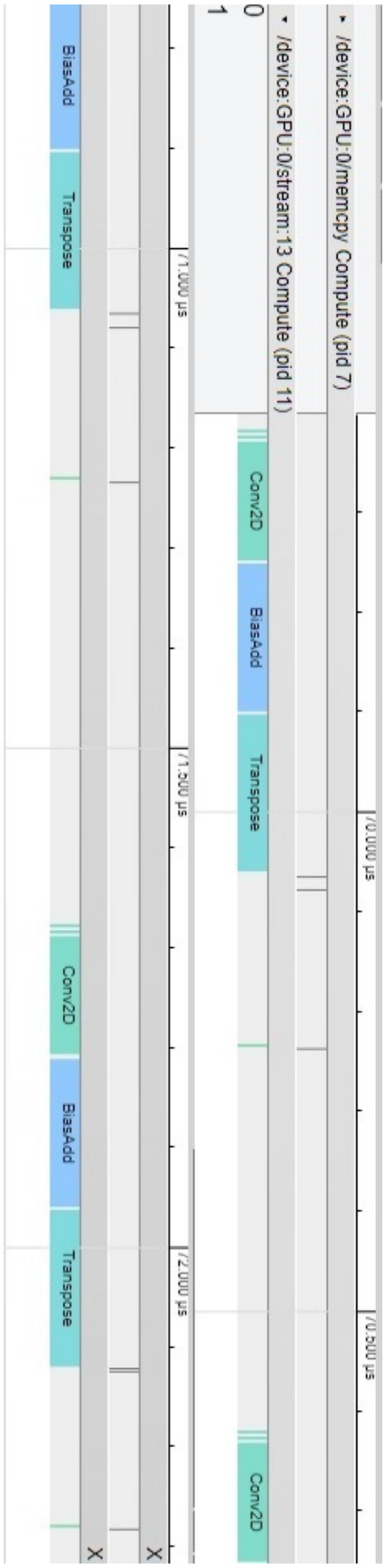

Figura 4.4: Comportamento do cálculo do *fitness* na camada convolucional da redes neural convolucional de 3 camadas com população de 40

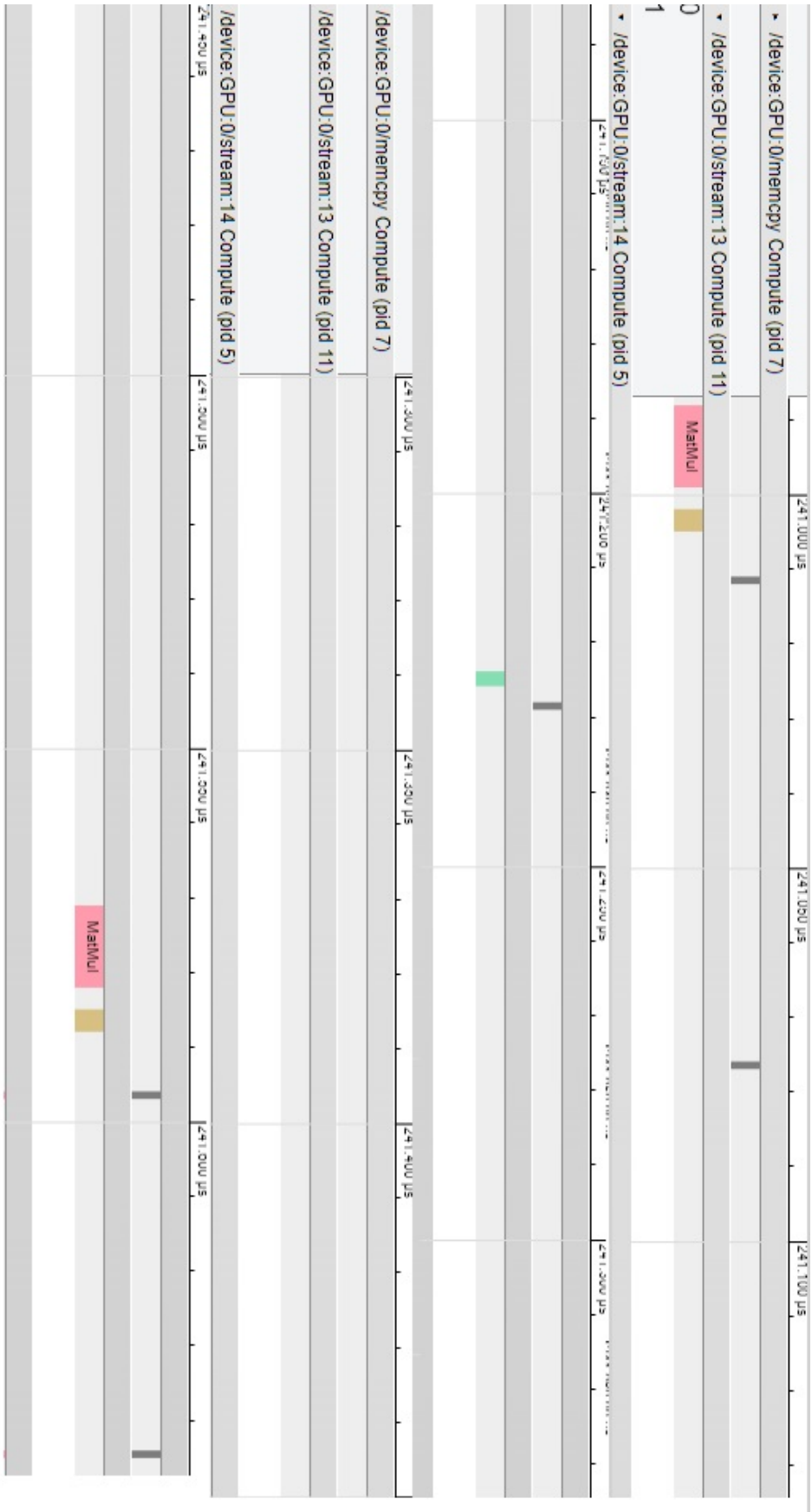

Figura 4.5: Comportamento do cálculo do *fitness* da redes neural convolucional na camada totalmente conectada de 3 camadas com população de 40

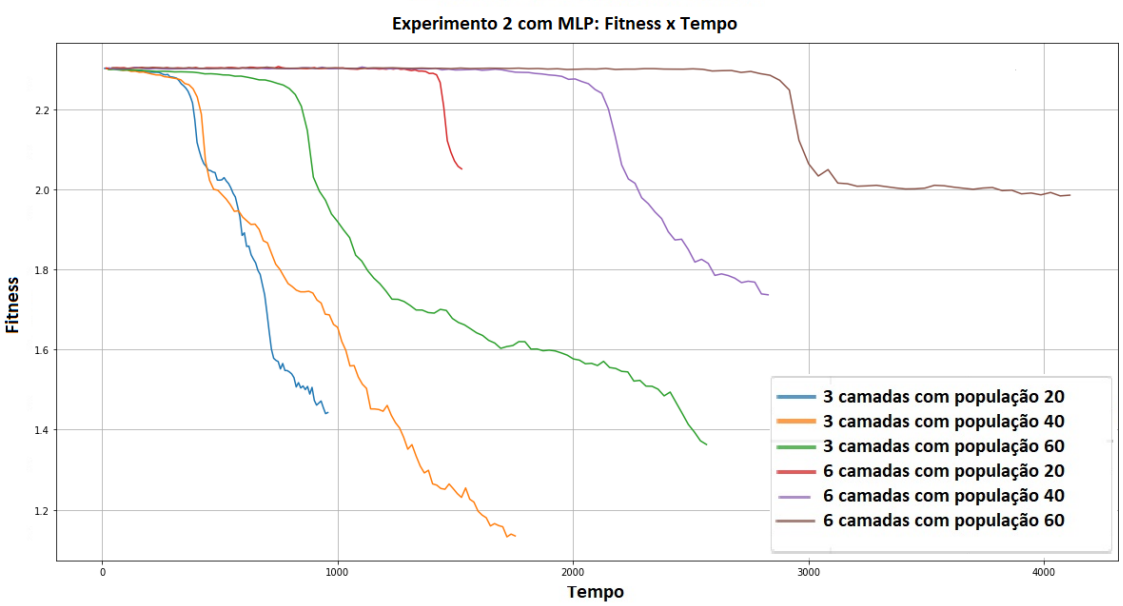

Figura 4.6: Melhor fitness das redes neurais *MLP* com função sigmoide com a estrutura proposta

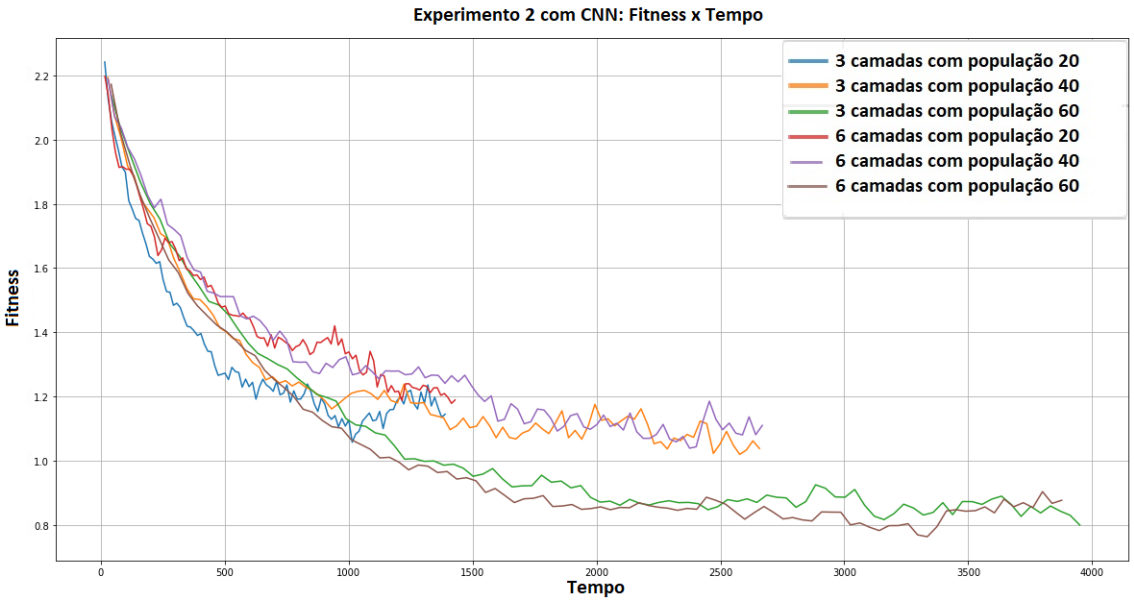

Figura 4.7: Melhor fitness das redes neurais convolucionais com função sigmoide com a estrutura proposta

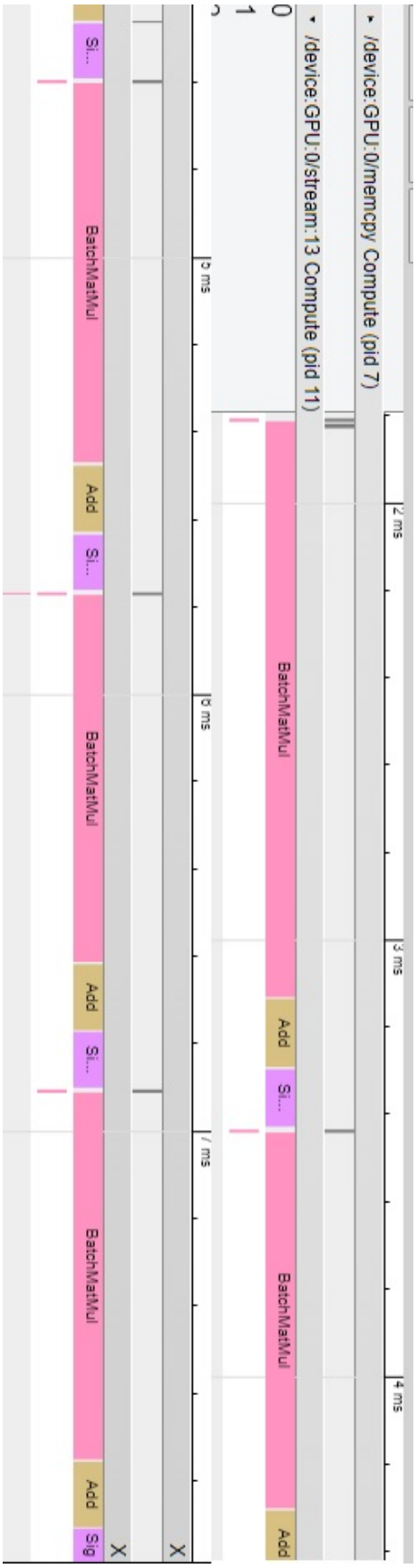

Figura 4.8: Comportamento do cálculo do fitness da redes neural *MLP* de 3 camadas com função sigmoide com população de 40

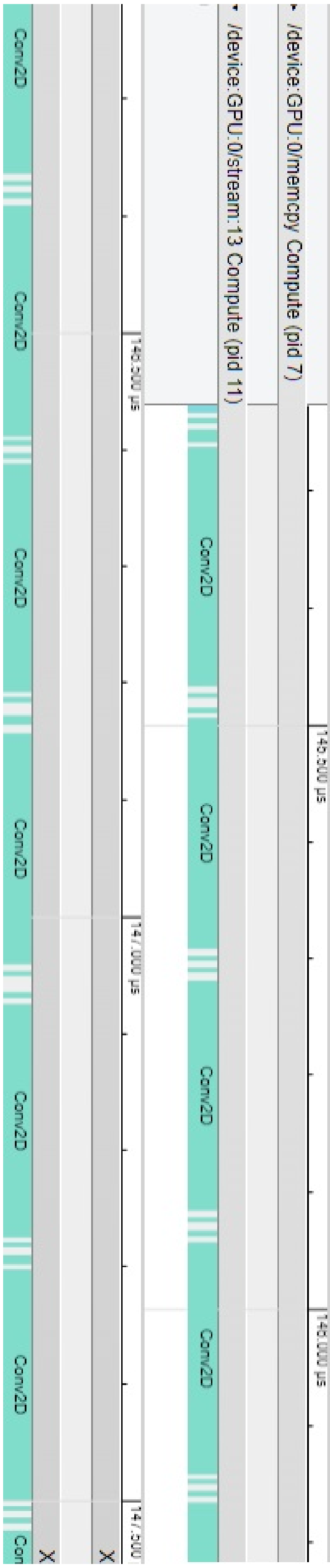

Figura 4.9: Comportamento do cálculo do fitness da redes neural convolucional de 3 camadas com função sigmoide com população de 40 nas camadas convolucionais

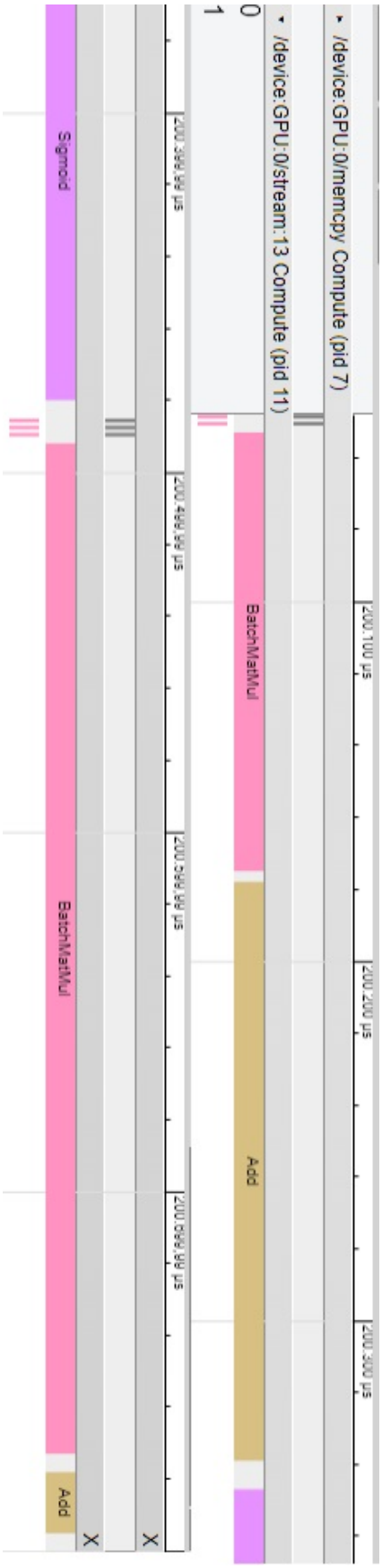

Figura 4.10: Comportamento do cálculo do fitness da redes neural convolucional de 3 camadas com função sigmoide com população de 40 nas camadas totalmente conectadas

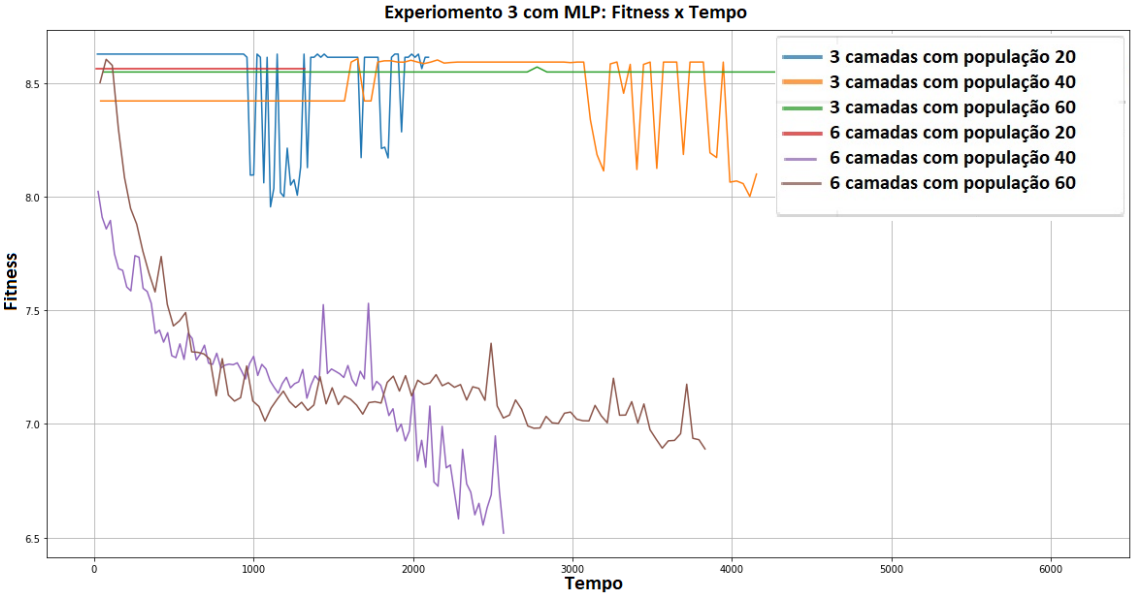

Figura 4.11: Melhor *fitness* das redes neurais *feedfoward* com função sigmoide utilizando sme

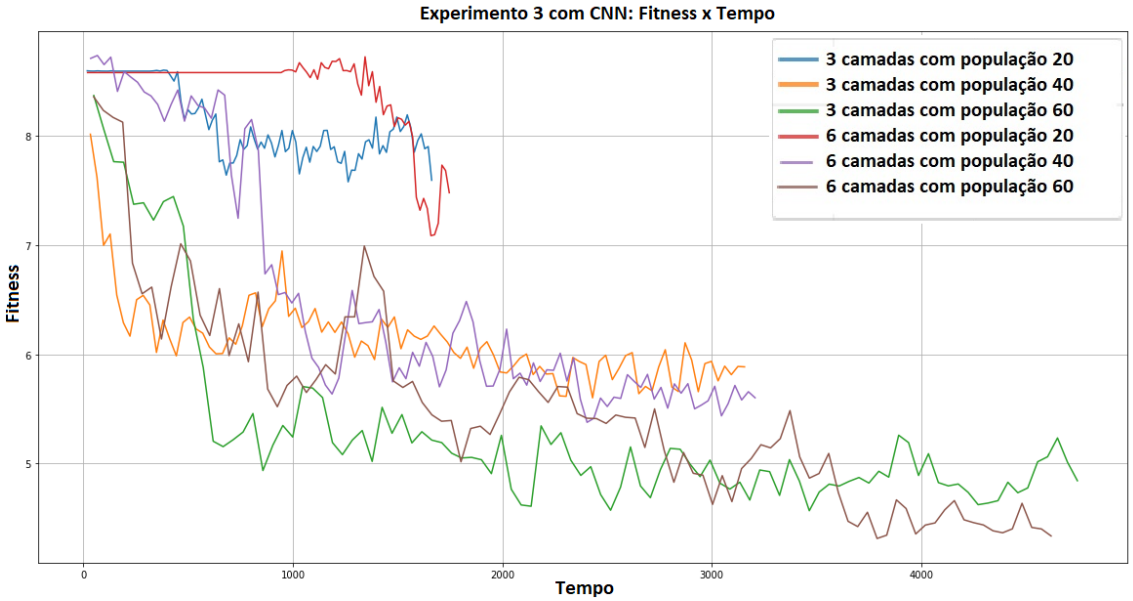

Figura 4.12: Melhor *fitness* das redes neurais convolucionais com função sigmoide utilizando SME

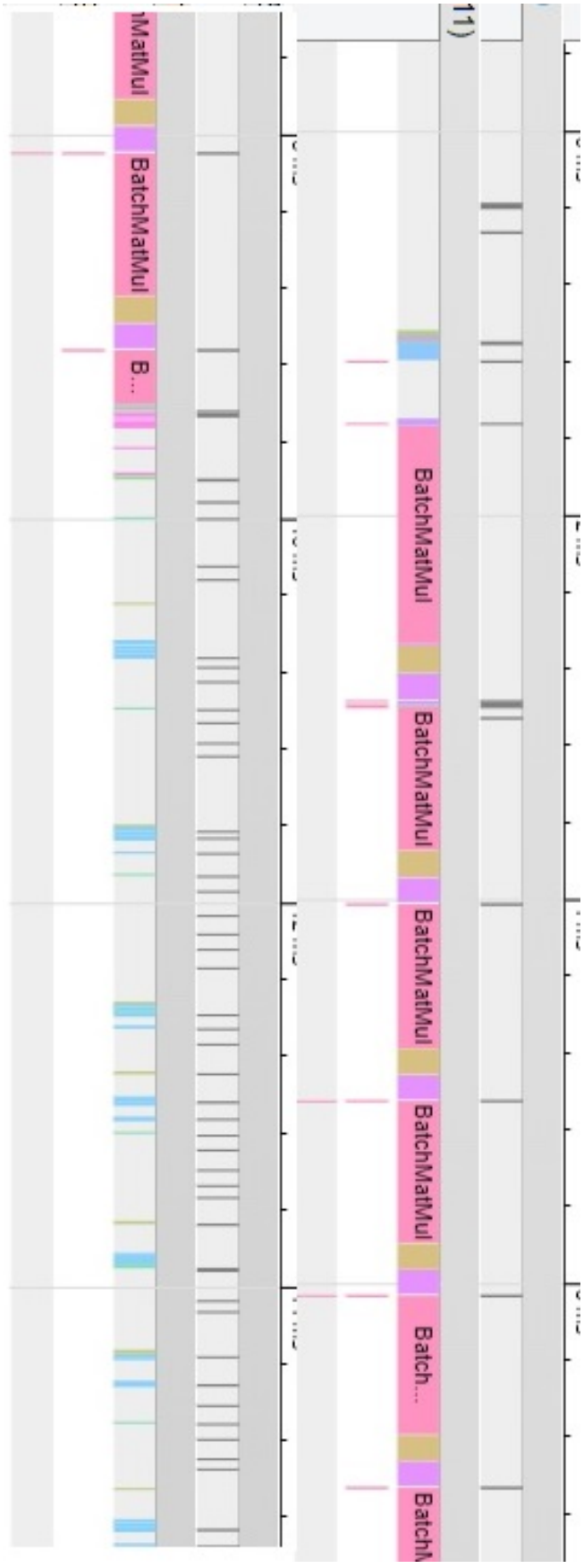

Figura 4.13: Comportamento da execução das redes neurais*MLP* com sme

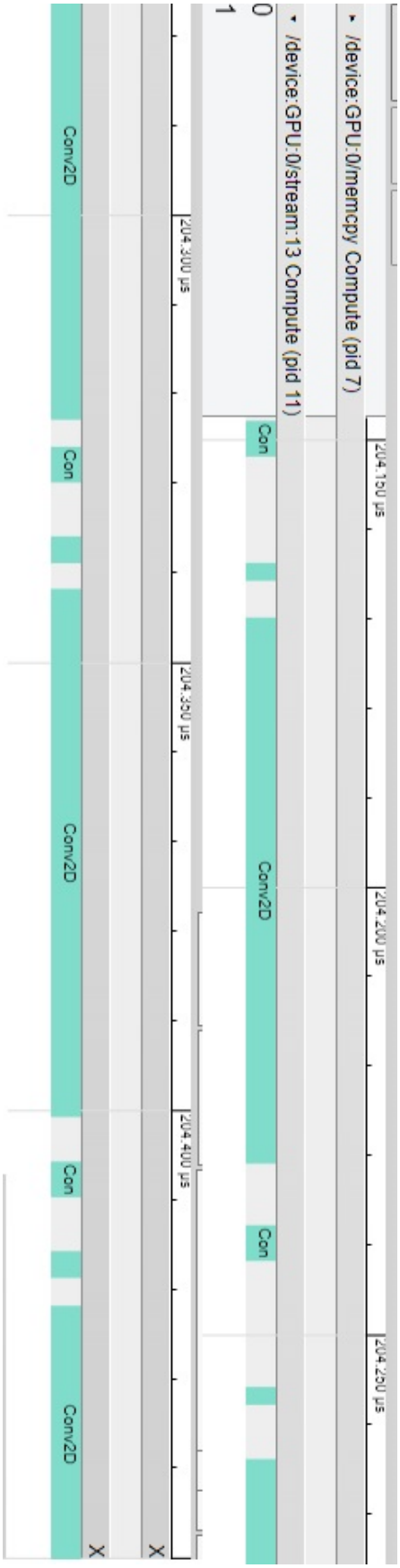

Figura 4.14: Comportamento da execução das camadas convolucionais nas redes neurais *CNN* com sme

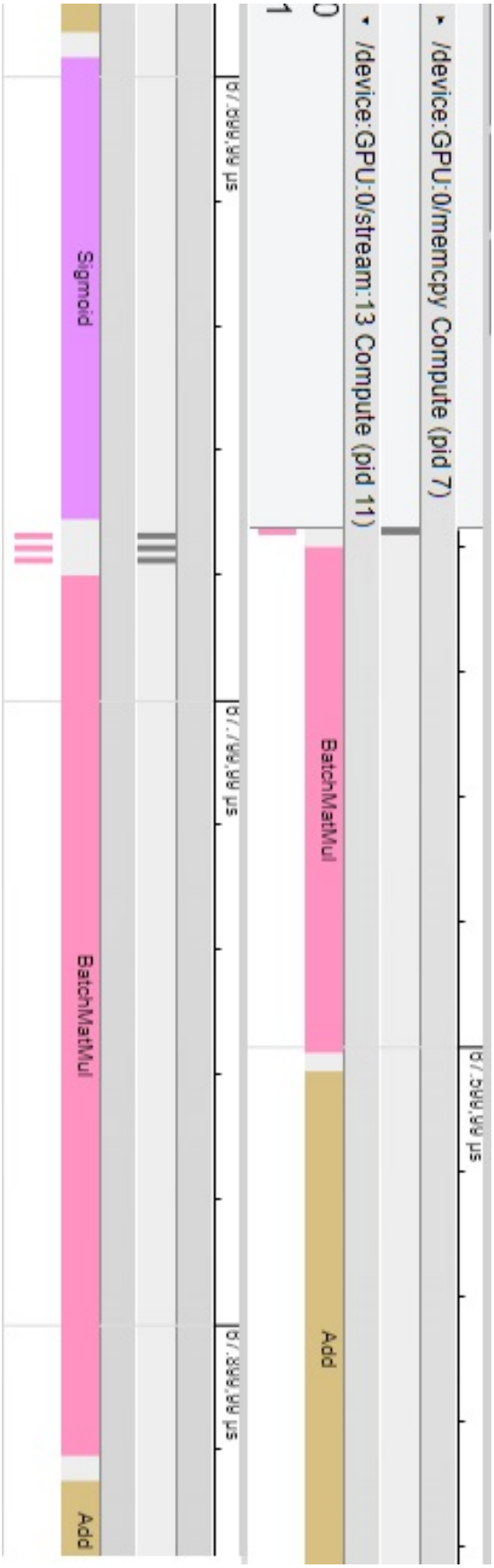

Figura 4.15: Comportamento da execução das camadas totalmente conectadas nas redes neurais *CNN* com sme

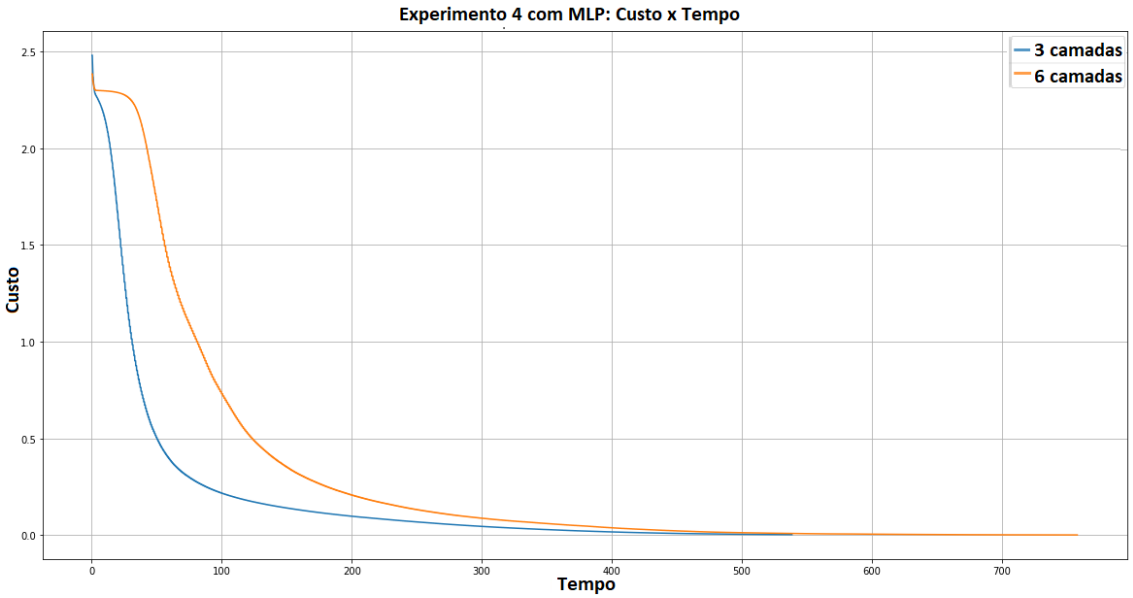

Figura 4.16: Custo das redes neurais *MLP* durante o treino utilizando ADAM

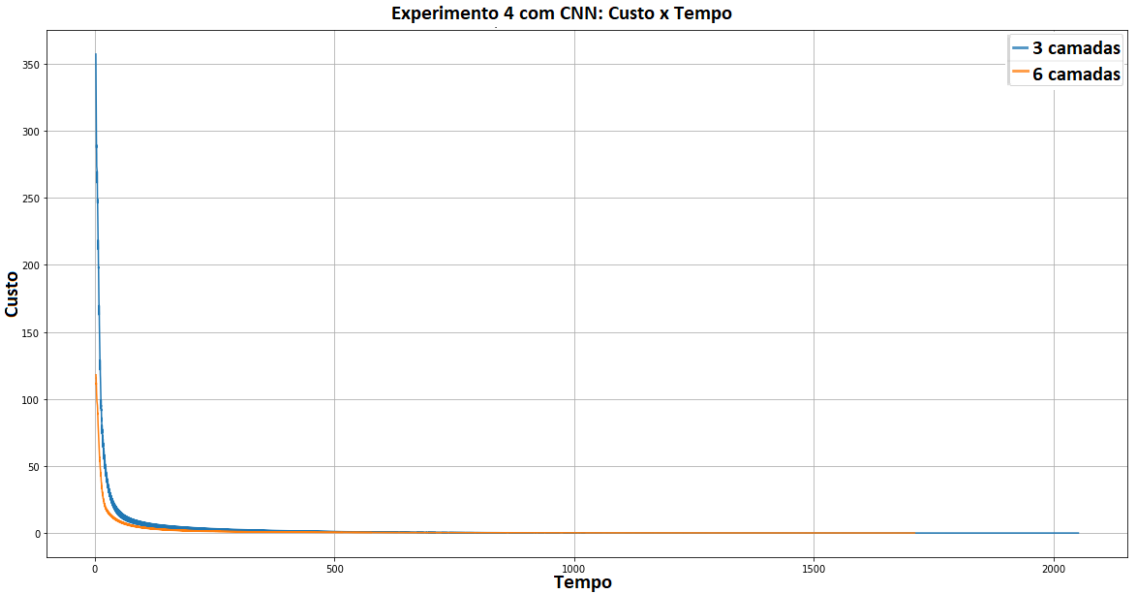

Figura 4.17: Custo das redes neurais convolucionais durante o treino utilizando ADAM

Capítulo 5

Conclusões

5.1 Considerações

De acordo com os resultados obtidos pelo experimento, somos capazes de afirmar que a estrutura proposta pelo trabalho contribuiu para acelerar a execução do algoritmo genético das redes neurais *MLP* e convolucionais. Apesar da execução algoritmo utilizando a estrutura não alcançar o mesmo *speedup* obtido nas redes neurais convolucionais em relação as *MLP*, é possível observar um pequeno ganho de velocidade de execução nas camadas totalmente conectadas. Embora o ganho não tenha sido em todas as etapas do algoritmo genético, limitando-se apenas ao cálculo do *fitness*, ainda há espaço para otimizações nas operações genéticas para utilizar a GPU junto com a estrutura proposta neste trabalho. Apesar disto, na comparação, o otimizador ADAM se demonstrou mais eficaz para o treino de redes neurais *MLP* e convolucionais para classificar imagens no *dataset* MNIST. O *workflow* proposto neste trabalho ajudou a a fazer um uso melhor das operações que utilizam a GPU, como podemos ver no resultado dos experimentos. Este workflow também ajudou a executar o algoritmo genético na GPU, principalmente na execução do cálculo do fitness, onde paralelizamos a execução de todas as camadas por vez, fazendo um uso menor dos recursos da GPU, em vez paralelizar a função fitness inteira, o que resultaria em uma alocação de recurso que não seria suportada na GPU que foi utilizada nos experimentos deste trabalho.

5.2 Contribuições

No treino dos pesos nas redes neurais evolutivas com a mesma estrutura, o trabalho conseguiu propor uma estrutura que otimize o desempenho na execução de múltiplas redes neurais sem prejudicar o desempenho da mesma. Já nas redes neurais *MLP*, conseguimos um ganho de desempenho de até 64% em relação ao tempo total de

execução, como um *speedup* acima de 20 na parte do cálculo do fitness.

A estrutura também contribuiu para uma pequena melhoria na execução das redes neurais convolucionais apesar de não ter não tido um ganho de velocidade de execução como na execução das camadas convolucionais, obteve um ganho para a execução das camadas totalmente conectadas que elas possuem.

5.3 Trabalhos futuros

Existem vários campos para explorar a paralelização do algoritmo utilizando esta estrutura, entre elas, melhorar o desempenho na execução de múltiplas camadas convolucionais no cálculo do fitness; melhorar a execução dos operadores genéticos que não obtiveram um ganho expressivo utilizando a estrutura; utilizar outros operadores genéticos não utilizados neste trabalho tendo em foco a paralelização que a estrutura proporciona. Outra possibilidade de trabalho futuro é ajustar esta estrutura para suportar redes neurais com estruturas diferentes, pois este trabalho só considerou a otimização de pesos onde a estrutura das redes na população é a mesma. Isto irá gerar alguns desafios, pois será difícil juntar os pesos em apenas um tensor de dimensão maior.

Referências Bibliográficas

[1] OWENS, J. D., HOUSTON, M., LUEBKE, D., et al. "GPU computing", 2008.

[2] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] SPECHT, D. F. "A general regression neural network", *IEEE transactions on neural networks*, v. 2, n. 6, pp. 568–576, 1991.

[4] WIDROW, B., LEHR, M. A. "30 years of adaptive neural networks: perceptron, madaline, and backpropagation", *Proceedings of the IEEE*, v. 78, n. 9, pp. 1415–1442, 1990.

[5] RUMELHART, D. E., HINTON, G. E., WILLIAMS, R. J. *Learning internal representations by error propagation*. Relatório técnico, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[6] KINGMA, D. P., BA, J. "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[7] YAO, X. "Evolving artificial neural networks", *Proceedings of the IEEE*, v. 87, n. 9, pp. 1423–1447, 1999.

[8] PALMES, P. P., HAYASAKA, T., USUI, S. "Mutation-based genetic neural network", *IEEE Transactions on Neural Networks*, v. 16, n. 3, pp. 587–600, 2005.

[9] HAUSKNECHT, M., LEHMAN, J., MIIKKULAINEN, R., et al. "A neuroevolution approach to general atari game playing", *IEEE Transactions on Computational Intelligence and AI in Games*, v. 6, n. 4, pp. 355–366, 2014.

[10] LAM, H., LING, S., LEUNG, F. H., et al. "Tuning of the structure and parameters of neural network using an improved genetic algorithm". In: *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No. 37243)*, v. 1, pp. 25–30. IEEE, 2001.

[11] MITCHELL, M. *An introduction to genetic algorithms*. MIT press, 1998.

[12] SANDERS, J., KANDROT, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[13] NVIDIA CORPORATION. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[14] DOLBEAU, R. "Theoretical peak FLOPS per instruction set: a tutorial", *The Journal of Supercomputing*, v. 74, n. 3, pp. 1341–1377, 2018.

[15] LOPEZ-RINCON, A., TONDA, A., ELATI, M., et al. "Evolutionary optimization of convolutional neural networks for cancer mirna biomarkers classification", *Applied Soft Computing*, v. 65, pp. 91–100, 2018.

[16] POSPICHAL, P., JAROS, J., SCHWARZ, J. "Parallel genetic algorithm on the cuda architecture". In: *European conference on the applications of evolutionary computation*, pp. 442–451. Springer, 2010.

[17] ZHANG, S., HE, Z. "Implementation of parallel genetic algorithm based on CUDA". In: *International Symposium on Intelligence Computation and Applications*, pp. 24–30. Springer, 2009.

[18] DEBATTISTI, S., MARLAT, N., MUSSI, L., et al. "Implementation of a simple genetic algorithm within the cuda architecture". In: *The Genetic and Evolutionary Computation Conference*, 2009.

[19] DALTON, S., OLSON, L., BELL, N. "Optimizing sparse matrix—matrix multiplication for the gpu", *ACM Transactions on Mathematical Software (TOMS)*, v. 41, n. 4, pp. 25, 2015.

[20] ABADI, M., BARHAM, P., CHEN, J., et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

[21] TREVOR, H., ROBERT, T., JH, F. "The elements of statistical learning: data mining, inference, and prediction". 2009.

[22] HAYKIN, S. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[23] MINSKY, M., PAPERT, S. A. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[24] GARLAND, M., LE GRAND, S., NICKOLLS, J., et al. "Parallel computing experiences with CUDA", *IEEE micro*, v. 28, n. 4, pp. 13–27, 2008.

[25] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[26] OBERMAYER, K., SCHULTEN, K., BLASDEL, G. *A neural network model for the formation and for the spatial structure of retinotopic maps, orientation-and ocular dominance columns.* University of Illinois at Urbana-Champaign, 1991.

[27] GLOROT, X., BORDES, A., BENGIO, Y. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

[28] PAINE, T., JIN, H., YANG, J., et al. "Gpu asynchronous stochastic gradient descent to speed up neural network training", *arXiv preprint arXiv:1312.6186*, 2013.

[29] PRECHELT, L. "Automatic early stopping using cross validation: quantifying the criteria", *Neural Networks*, v. 11, n. 4, pp. 761–767, 1998.

[30] GOLDBERG, D. E. "Genetic algorithms in search, optimization, and machine learning", 1989.

[31] MONTANA, D. J., DAVIS, L. "Training Feedforward Neural Networks Using Genetic Algorithms." In: *IJCAI*, v. 89, pp. 762–767, 1989.

[32] STANLEY, K. O., MIIKKULAINEN, R. "Efficient evolution of neural network topologies". In: *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, v. 2, pp. 1757–1762. IEEE, 2002.

[33] KUMAR, M., HUSIAN, M., UPRETI, N., et al. "Genetic algorithm: Review and application", *International Journal of Information Technology and Knowledge Management*, v. 2, n. 2, pp. 451–454, 2010.

[34] MURATA, T., ISHIBUCHI, H., TANAKA, H. "Multi-objective genetic algorithm and its applications to flowshop scheduling", *Computers & Industrial Engineering*, v. 30, n. 4, pp. 957–968, 1996.

[35] HALL, L., BEZDEK, J., BOGGAVARPU, S., et al. "Genetic fuzzy clustering". In: *NAFIPS/IFIS/NASA'94. Proceedings of the First International Joint Conference of The North American Fuzzy Information Processing Society*

Biannual Conference. The Industrial Fuzzy Control and Intellige, pp. 411–415. IEEE, 1994.

[36] BACK, T. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[37] ZHONG, J., HU, X., ZHANG, J., et al. "Comparison of performance between different selection strategies on simple genetic algorithms". In: *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, v. 2, pp. 1115–1121. IEEE, 2005.

[38] RAZALI, N. M., GERAGHTY, J., OTHERS. "Genetic algorithm performance with different selection strategies in solving TSP". In: *Proceedings of the world congress on engineering*, v. 2, pp. 1–6. International Association of Engineers Hong Kong, 2011.

[39] SASTRY, K., GOLDBERG, D., KENDALL, G. "Genetic Algorithms". In: Burke, E. K., Kendall, G. (Eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pp. 97–125, Boston, MA, Springer US, 2005. ISBN: 978-0-387-28356-2. doi: 10.1007/0-387-28356-0_4. Disponível em: <https://doi.org/10.1007/0-387-28356-0_4>.

[40] VASUDEVAN, M., TIAN, Y.-C., TANG, M., et al. "Using genetic algorithm in profile-based assignment of applications to virtual machines for greener data centers". In: *International Conference on Neural Information Processing*, pp. 182–189. Springer, 2015.

[41] DE JONG, K. "Adaptive system design: a genetic approach", *IEEE Transactions on Systems, Man, and Cybernetics*, v. 10, n. 9, pp. 566–574, 1980.

[42] STANLEY, K. O., MIIKKULAINEN, R. "Evolving neural networks through augmenting topologies", *Evolutionary computation*, v. 10, n. 2, pp. 99–127, 2002.

[43] DING, S., LI, H., SU, C., et al. "Evolutionary artificial neural networks: a review", *Artificial Intelligence Review*, v. 39, n. 3, pp. 251–260, Mar 2013. ISSN: 1573-7462. doi: 10.1007/s10462-011-9270-6. Disponível em: <https://doi.org/10.1007/s10462-011-9270-6>.

[44] HECKERLING, P. S., GERBER, B. S., TAPE, T. G., et al. "Use of genetic algorithms for neural networks to predict community-acquired pneumonia", *Artificial Intelligence in Medicine*, v. 30, n. 1, pp. 71–84, 2004.

[45] DZIEKONSKI, A., LAMECKI, A., MROZOWSKI, M. "A memory efficient and fast sparse matrix vector product on a GPU", *Progress In Electromagnetics Research*, v. 116, pp. 49–63, 2011.

[46] MEI, X., CHU, X. "Dissecting GPU Memory Hierarchy Through Microbenchmarking", *IEEE Transactions on Parallel and Distributed Systems*, v. 28, n. 1, pp. 72–86, Jan 2017. ISSN: 1045-9219. doi: 10.1109/TPDS.2016.2549523.

[47] LECUN, Y., CORTES, C. "MNIST handwritten digit database", 2010. Disponível em: <`http://yann.lecun.com/exdb/mnist/`>.

[48] HE, K., ZHANG, X., REN, S., et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[49] CLEVERT, D.-A., UNTERTHINER, T., HOCHREITER, S. "Fast and accurate deep network learning by exponential linear units (elus)", *arXiv preprint arXiv:1511.07289*, 2015.

[50] HE, K., ZHANG, X., REN, S., et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[51] HUANG, G., LIU, Z., VAN DER MAATEN, L., et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[52] SIMARD, P. Y., STEINKRAUS, D., PLATT, J. C., et al. "Best practices for convolutional neural networks applied to visual document analysis." In: *Icdar*, v. 3, 2003.

[53] MIKOLOV, T., KOMBRINK, S., BURGET, L., et al. "Extensions of recurrent neural network language model". In: *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5528–5531. IEEE, 2011.

[54] HINTON, G., VINYALS, O., DEAN, J. "Distilling the knowledge in a neural network", *arXiv preprint arXiv:1503.02531*, 2015.

[55] GOLIK, P., DOETSCH, P., NEY, H. "Cross-entropy vs. squared error training: a theoretical and experimental comparison." In: *Interspeech*, v. 13, pp. 1756–1760, 2013.

[56] SAINATH, T. N., MOHAMED, A.-R., KINGSBURY, B., et al. "Deep convolutional neural networks for LVCSR". In: *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8614–8618. IEEE, 2013.

[57] MARTÍNEZ-ESTUDILLO, F. J., HERVÁS-MARTÍNEZ, C., GUTIÉRREZ, P. A., et al. "Evolutionary product-unit neural networks classifiers", *Neurocomputing*, v. 72, n. 1-3, pp. 548–561, 2008.

[58] ABADI, M., AGARWAL, A., BARHAM, P., et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". 2015. Disponível em: <`https://www.tensorflow.org/`>. Software available from tensorflow.org.

[59] HUNTER, J. D. "Matplotlib: A 2D graphics environment", *Computing in Science & Engineering*, v. 9, n. 3, pp. 90–95, 2007. doi: 10.1109/MCSE.2007.55.

Apêndice A

Imagens

A.0.1 experimento 1: *feedfoward* simples

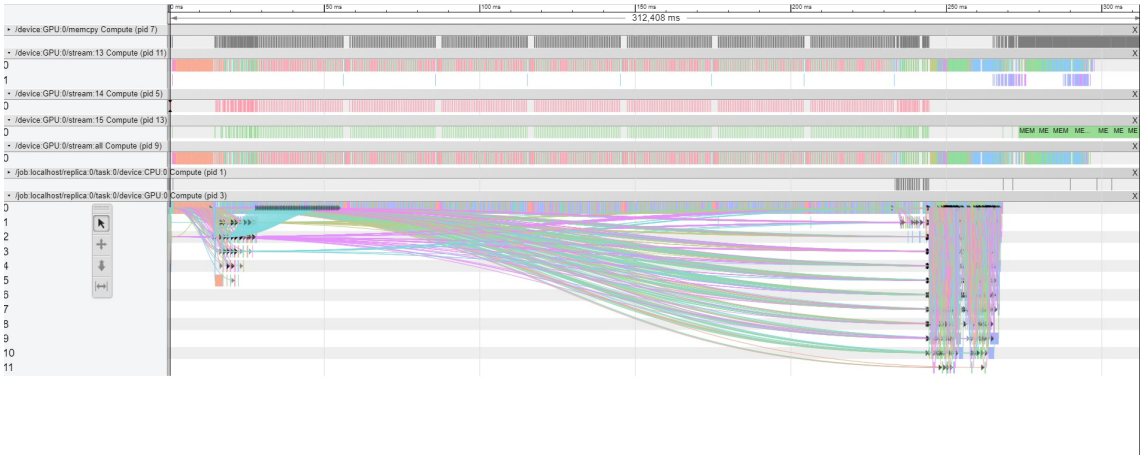

Figura A.1: Comportamento da execução das redes neurais *feedfoward* simples no experimento 1

A.0.2 experimento 1: convolucional

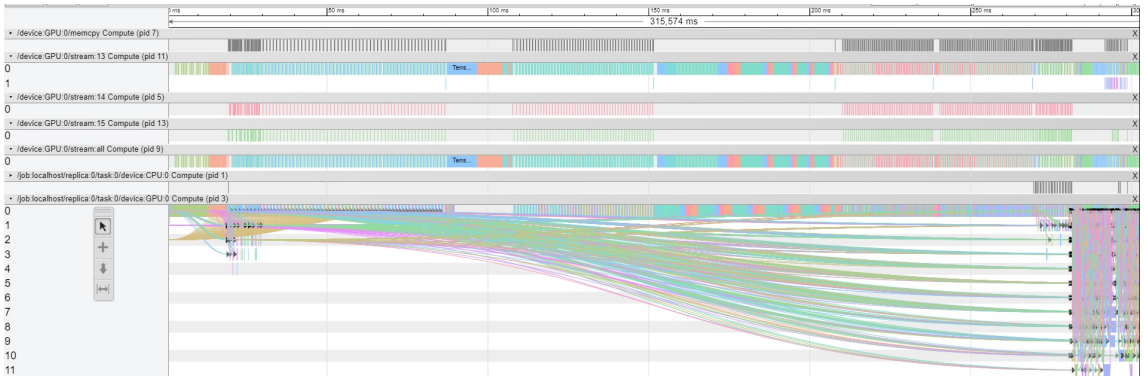

Figura A.2: Comportamento da execução das redes neurais convolucionais simples no experimento 1