
# Optimização de Redes Neuronais Convolucionais

## Projeto II

Tiago João Amaral Martins

**Orientadores**

Ana Paula Neves Ferreira da Silva

Arlindo Ferreira da Silva

Trabalho de Projeto apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Licenciado em Engenharia Informática, realizada sob a orientação científica do Professor Adjunto Doutor Arlindo Ferreira da Silva e coorientação da Professora Adjunta Doutora Ana Paula Neves Ferreira da Silva, do Instituto Politécnico de Castelo Branco.

**Junho de 2024**

## Composição do júri

Presidente do júri

Doutor, João Manuel Leitão Pires Caldeira

Vogais

Doutor, Arlindo Ferreira da Silva

Professor Adjunto da Escola Superior de Tecnologia de Castelo Branco

Doutor, Fernando Reinaldo da Silva Garcia Ribeiro

Professor Adjunto da Escola Superior de Tecnologia de Castelo Branco

# Agradecimentos

Expresso os meus sinceros agradecimentos ao Professor Arlindo Ferreira da Silva e à Professora Ana Paula Neves Ferreira da Silva pela inestimável assistência prestada. Agradeço-lhes por estarem constantemente disponíveis para esclarecer quaisquer dúvidas que pudessem surgir, e, sobretudo, pelo valioso *feedback* e pelos conselhos providenciados com o propósito de assegurar a excelência e qualidade do projeto.

## Resumo

Este relatório foi elaborado no âmbito da disciplina da Unidade Curricular de Projeto I/II, do segundo semestre da Licenciatura de Engenharia Informática, curso este lecionado na Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco. O projeto visa explorar e entender algoritmos que permitam encontrar arquiteturas de redes neuronais convolucionais que executem no menor tempo possível e necessitem do mínimo de recursos, pois serão executados em hardware limitado. Para alcançar esses objetivos, fez-se uma review sistemática da literatura, para assim se obterem os artigos do estado da arte referentes a propostas de algoritmos que pretendem minimizar o tempo de execução, maximizando a qualidade das arquiteturas encontradas. Os algoritmos devem ser evolucionários, pois estes são populares para resolver problemas com vastos espaços de possibilidades. Após analisar, individualmente, cada algoritmo, é feita uma análise geral, de maneira a identificar quais as melhores metodologias e estratégias para se alcançar os objetivos propostos. É descrita a implementação de um caso de estudo que procura ajudar a definir o tipo de algoritmo e estratégia a serem utilizados na segunda fase deste projeto.

Na conclusão do Projeto I, encontraram-se duas métricas que estimam a qualidade de uma arquitetura de rede neuronal convolucional sem fazer qualquer tipo de treino. Em continuação, na segunda fase do projeto, integraram-se estas métricas em algoritmos de programação genética que, ao contrário daqueles encontrados no estado da arte no Projeto I, utilizam representações complexas como gramáticas e grafos. Adaptou-se o código destes algoritmos para utilizarem estas duas métricas em *Tensorflow*. Posteriormente, obtiveram-se os resultados através de um treino intensivo das melhores arquiteturas encontradas. Constatou-se que o estado da arte foi superado ao nível de tempo de procura e obteve-se exatidões próximas do estado da arte e superiores aos algoritmos originais de programação genética. Concluiu-se também que a melhor combinação é a métrica gananciosa, NASWOT, com o algoritmo CGP-CNN.



## Palavras-chave

Redes neuronais convolucionais

Procura de arquitetura de redes neuronais

Otimização de arquitetura de redes neuronais

Algoritmos evolutivos

Deep Learning

# Abstract

This report was written as part of the Project I/II curricular unit of the second semester of the Computer Engineering degree, taught at Escola Superior de Tecnologia of Instituto Politécnico de Castelo Branco. The project aims to explore and understand algorithms that enable the discovery of convolutional neural network architectures that run in the shortest possible time and require the least resources, as they will run on limited hardware. To achieve these objectives, we conducted a systematic literature review to obtain up-to-date articles on proposed algorithms that aim to minimize execution time while maximizing the quality of the architectures found. The algorithms must be evolutionary, as they are popular for solving problems with vast spaces of possibilities. After analyzing each algorithm individually, a general analysis will be conducted to outline the best methodologies and strategies for achieving the proposed objectives. Then, is done the implementation of a case study to help define the type of algorithm and strategy to be used in the second phase of this project.

In the conclusion of Project I, which found two metrics that estimate the quality of a convolutional neural network architecture without any training. In this second phase, these metrics were implemented in genetic programming algorithms, which, unlike the state of the art in Project I, have complex representations such as grammars and graphs. The code of these algorithms was adapted to use the implementation of these metrics in Tensorflow, and the results were then extracted through intensive training of the best architectures found. It was found that the search time was better than the state of the art and the accuracy was close to the state of the art and better than the original genetic programming algorithms. It was also concluded that the best combination is the greedy metric, NASWOT, with the CGP-CNN algorithm.



# Keywords

Convolutional neural networks

Neural architecture search

Deep Learning

Evolutionary algorithms

Artificial intelligence

# Índice geral

# Índice de figuras

# Lista de tabelas

## Lista de abreviaturas, siglas e acrónimos

ACO – *Ant Colony Optimization*

API – *Application Programming Interface*

AUC – *Area Under Curve*

CDBN – *Convolutional DBN*

CGP - *Cartesian Genetic Programming*

CRBM – *Convolutional RBM*

DAG – *Directed Acyclic Graph*

DBN – *Deep Belief Network*

DENSER - *Deep Evolutionary Network Structured Representation*

FLOPS – *Floating-point Operations Per Second*

GP-CNAS - *Genetic Programming-Convolutional Neural Architecture Search*

GPU – *Graphics Processing Unit*

IA – Inteligência Artificial

ILSVRC - *ImageNet Large Scale Visual Recognition Challenge*

MLP – *MultiLayer Perceptron*

MSE – *Mean Squared Error*

NAS – *Neural Architecture Search*

NASWOT - *Neural Architecture Search Without Training*

NTK - *Neural Tangent Kernel*

PSO – *Particle Swarm Optimization*

RBM – *Restricted Boltzmann Machine*

ReLU – *Rectified Linear Unit*

RGB – *Red Green Blue*

ROC - *Receiver Operating Characteristic*

SGD – *Stochastic Gradient Descent*

SVM – *Support Vector Machine*

TFLOPS – *Tera* FLOPS

TPU – *Tensor Processing Unit*

# 1. Introdução

Redes neuronais convolucionais são modelos matemáticos que permitem extrair e aprender informação de imagens ou de outros sinais [1]. Diversas topologias foram criadas, como a *AlexNet* [2], VGG [3], *Inception* [4], *ResNet* [5] e *DenseNet* [6], onde todas utilizam convoluções, mas cada uma aplica topologias distintas, permitindo explorar diferentes comportamentos.

No entanto, criar topologias para estas redes requer sabedoria, experiência e diversos testes, impossibilitando a criação de novas topologias para aqueles com menos experiência e conhecimento.

Algoritmos evolucionários são utilizados para procurar arquiteturas, automaticamente, para estas redes. Um dos mais populares é o [7] que faz uso de aprendizagem por reforço para encontrar as arquiteturas. Contudo, este tipo de aprendizagem exige demasiados recursos e por isso, optou-se por explorar neste projeto vários tipos de algoritmos evolucionários, já que estes são mais indicados para grandes conjuntos de possibilidades. Estes algoritmos foram evoluindo para consumir cada vez menos recursos e obter melhores arquiteturas. Consequentemente, foram surgindo na literatura diversas estratégias para abordar este problema, algoritmos e representações, incluindo algoritmos genéticos, programação genética e inteligência por enxame como mencionado em [8].

Porém, todos estes algoritmos procuram arquiteturas de redes neuronais em vastos espaços de possíveis soluções. Logo, tempo e recursos continuam a constituir uma limitação significativa, dado que estes algoritmos necessitam de treinar as redes neuronais para as avaliar, exigindo assim recursos computacionais elevados para executar os treinos. Quando há recursos "ilimitados", isto não é um problema, mas nem sempre é possível garantir os recursos necessários.

Considerando estes dois problemas, é necessário desenvolver algoritmos que procurem minimizar o tempo e os recursos utilizados para a procura de arquiteturas de redes neuronais convolucionais. Algumas metodologias são cada vez mais exploradas, como a utilização de modelos *surrogate* [9] ou não fazer qualquer treino das redes como em [10] e [11].

## 1.1. Enquadramento

Como a exploração de metodologias mais eficientes para a procura de arquiteturas ainda é algo recente na literatura, na primeira parte do projeto, examinou-se o estado da arte e analisou-se artigos que implementam, de forma eficiente, a procura de arquiteturas de redes neuronais com algoritmos evolucionários.

Posteriormente, com os resultados obtidos, traçou-se uma direção favorável ao desenvolvimento de algoritmos evolucionários que garantam a qualidade das soluções,

mantendo um tempo de procura reduzido em um ambiente com recursos limitados, por exemplo, ambientes que utilizem apenas uma GPU (*Graphics Processing Unit*).

Por fim, na segunda parte do projeto, considerando o resultado da análise e sendo identificadas abordagens que permitam avaliar uma arquitetura sem treinar a rede, ou, pelo menos, que estimem essa avaliação de forma computacionalmente pouco dispendiosa, são implementados e testados diversos algoritmos que têm essa característica, de forma a verificar quais os melhores, nos diversos datasets e quais as alterações que podem ser feitas de forma a melhorar os resultados.

## 1.2. Objetivos

Numa primeira fase, foi feita uma revisão sistemática do estado da arte referente a algoritmos que pretendem minimizar o tempo de execução e/ou são executados em *hardware* limitado. A definição de *hardware* limitado será a utilização de apenas uma GPU. Após as primeiras etapas da revisão (seleção, exclusão e resumo de artigos), foi elaborada uma análise dos resultados, podendo-se assim definir quais as melhores técnicas para alcançar a minimização do tempo e utilização de recursos. Por fim, foram descritas algumas implementações de algoritmos para complementar a análise com outros resultados.

Após a identificação das melhores técnicas, nesta segunda fase do projeto, foram implementados algoritmos que as utilizem, mais especificamente, algoritmos de programação genética, pois têm representações mais complexa tendendo a demorar mais a executar a procura. Assim poder-se-á constatar se as técnicas encontradas preservam os resultados, mas minimizam o tempo, resultando em algoritmos mais robustos na exploração do espaço de procura.

## 1.3. Planeamento e organização do projeto

Para o desenvolvimento da primeira fase deste projeto, despendeu-se grande parte do tempo na procura e análise de artigos a incluir na revisão sistemática, já que esta fundamenta todo o projeto. Foram ainda implementados alguns dos algoritmos e/ou metodologias encontradas na revisão, para assim se complementar a análise geral e definir as metodologias que devem ser consideradas para a segunda fase.

Na segunda fase do projeto, utilizar-se-ão duas métricas encontradas durante a revisão sistemática, sendo estas o NASWOT e o NTK. Estas serão aplicadas a algoritmos de programação genética que utilizam representações complexas e permitem uma evolução mais livre. As arquiteturas de redes neuronais convolucionais obtidas, serão testadas nos *datasets* mais utilizados pelos autores dos diversos artigos encontrados na revisão sistemática. Estes *datasets* são o CIFAR-10 e o CIFAR-100, em que o primeiro é mais simples e menos desafiador que o último. Os testes serão executados num ambiente com baixos recursos computacionais, tendo apenas uma GPU.

Este relatório documenta a totalidade do trabalho realizado no projeto, com o objetivo de ser autocontido e não obrigar a consulta do relatório anterior. Neste sentido, os capítulos 2,3 e 4 transitam sem alterações do relatório de projeto I, os capítulos 1e 8 sofreram alterações e foram acrescentados os capítulos 5, 6 e 7.

Este relatório de projeto está assim organizado em 8 capítulos:

- O primeiro capítulo introduz o problema da procura de arquiteturas de redes neuronais convolucionais e como será executada uma revisão sistemática e a implementação de algoritmos de programação genética que utilizem as métricas identificadas na revisão.
- O segundo capítulo descreve o que é inteligência artificial, introduzindo diversos campos da área. No fim deste capítulo, o leitor deverá ter um conhecimento básico sobre redes neuronais e algoritmos evolucionários.
- O terceiro capítulo mostra todo o processo da revisão sistemática, onde constam os resumos dos artigos selecionados, uma análise geral e a descrição dos *datasets* encontrados, além da exploração de metodologias similares que podem complementar a revisão.
- O quarto capítulo apresenta um caso de estudo, onde se implementa um dos algoritmos apresentados no capítulo três, verificando se o mesmo deve ser utilizado na segunda fase do projeto.
- O quinto capítulo inicia a segunda fase do projeto, onde se relata toda a adaptação do código original dos algoritmos de programação genética selecionados, para utilizarem as métricas identificadas na revisão sistemática.
- O sexto capítulo apresenta os resultados das implementações, concluindo-se qual a melhor combinação para obter resultados satisfatórios nos *datasets* selecionados
- O sétimo capítulo apresenta uma breve discussão sobre os resultados e as decisões tomadas para tentar melhorá-los. Apresentam-se novas possibilidades de melhoria.
- O oitavo e último capítulo concluem a segunda e última do projeto, constatando tudo o que foi feito

# 2. Inteligência Artificial

A inteligência artificial é uma área científica que foi criada após a segunda guerra mundial, tendo o seu nome sido cunhado em 1956 por *John McCarthy* [12]. É difícil encontrar um significado preciso para inteligência artificial, já que também não compreendemos completamente o que é inteligência num contexto mais geral. No entanto, há quatro abordagens distintas à tentativa de definir inteligência artificial [12]: pensar como um humano, agir como um humano, pensar racionalmente e agir racionalmente. Para proporcionar um maior entendimento sobre cada uma destas definições, abordar-se-á cada uma detalhadamente nos próximos parágrafos.

Começando pela definição "pensar como um humano", esta implica a necessidade de entender como nós pensamos. Inclui, normalmente, experiências com seres humanos e o estudo da reação do cérebro em diversas tarefas. Considera-se que uma IA (Inteligência Artificial) pensa como um humano se, ao apresentar-se a ambos um mesmo problema, ela seguir o mesmo caminho que o ser humano seguiu para o resolver. A ciência cognitiva combina IA com o resultado destas experiências e tem sido aplicação, por exemplo, na área da visão computacional, pois vários modelos são baseados na neurofisiologia.

"Agir como um humano" é algo que foi estudado em 1950 por *Alan Turing*, autor do teste de *Turing*. O teste consiste em um examinador fazer um conjunto de perguntas simultaneamente a um computador e a um humano, os quais não estão visíveis. Se o entrevistador identificar incorretamente a máquina como sendo o humano, aquela passaria o teste de *Turing*. Todavia, apesar de ainda ser relevante, poucos cientistas se focam atualmente em tentar fazer as IA passarem neste teste, pois é preferível estudar e entender os princípios de como a inteligência funciona do que apenas garantir apenas a imitação de comportamentos humanos.

O "pensamento racional" é algo que foi introduzido pelo filósofo Aristóteles, baseando-se na criação de sequências simbólicas e formais que, ao serem produzidas com base em premissas verdadeiras, devolvem também um resultado verdadeiro. Esta forma de pensamento racional denominada de lógica, foi implementada computacionalmente em várias formas. Esta implementação encontrou diversos obstáculos, como, por exemplo, premissas que nem sempre são verdadeiras não podem ser utilizadas na lógica, porém estas são comuns no mundo real e é necessário obter respostas de igual forma. A próxima definição permite ultrapassar alguns destes obstáculos.

A abordagem do "agir racionalmente" introduz uma entidade denominada de agente, i.e., algo que faz alguma ação. Nesta definição, o agente não se baseia apenas na lógica pura, referenciada acima, pois esta nem sempre funciona. Um agente observa o ambiente e adapta-se de forma a obter o melhor resultado possível face ao que o ambiente lhe apresenta. Mesmo assim, nem sempre é possível ao agente tomar a decisão mais racional. *Gödel* propôs o teorema da incompletude, concluindo que há certas funções que não podem ser calculadas por uma máquina, embora seja possível

aproximá-las, utilizando a teoria das probabilidades e outros ramos da matemática. A junção das aproximações imita o comportamento humano, já que os humanos se ajustam a qualquer situação, mantendo a sua racionalidade. Esta será a definição que melhor se aproxima dos campos da IA que serão introduzidos nos próximos subcapítulos.

Agora que foram descritas as várias formas de definir IA, pode-se apresentar algumas descobertas históricas que fizeram parte da evolução que permitiu a criação e melhoria dos diversos campos que serão estudados e/ou utilizados neste projeto [12].

Tudo começou em 1943 com *Warren McCulloch* e *Walter Pitts* que, ao fazerem utilização de estudos de neurofisiologia, lógica e de teoria da computação, criaram a primeira e primitiva rede neuronal artificial, onde um conjunto de neurónios era capaz simular operações lógicas [13]. Os neurónios podem ser ativados e desativados considerando se os neurónios vizinhos estão igualmente estimulados ou não. Assim, em 1949, *Donald Hebb* constatou que é possível modificar a força das conexões entre neurónios, sendo a forma de o fazer posteriormente denominada como regra de *Hebb* [14]. Na Figura 1 pode observar-se a estrutura de um neurónio biológico, o qual é imitado pelos neurónios artificiais: as dendrites recebem algum tipo de entrada, estas entradas são processadas no corpo do neurónio (*soma*) e o resultado é enviado para o axónio, que por sua vez se vai ligar às dendrites de outros neurónios. Considerando o resultado do processamento, pode ou não haver ativação do neurónio e transmissão do sinal ao longo do axónio.

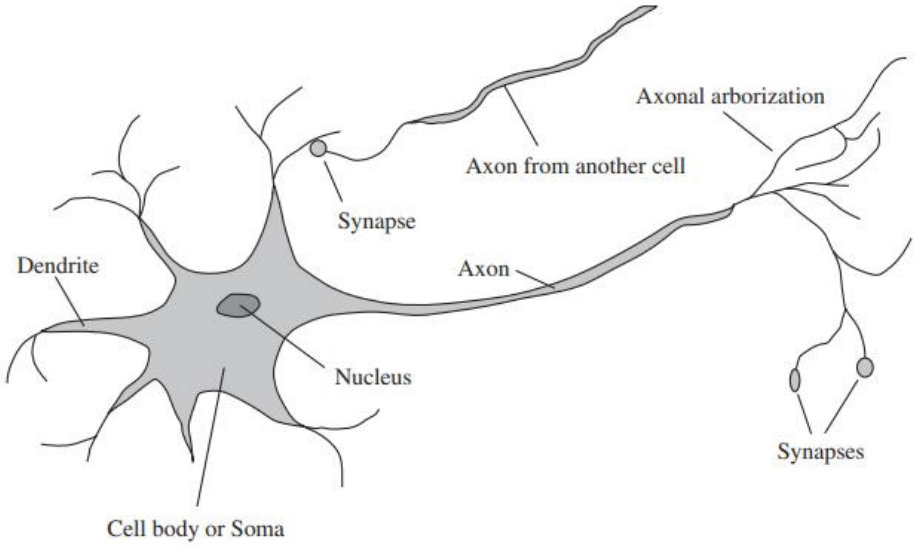


Figura 1 - Representação de um neurónio [12]

Em 1956, *Newell* e *Simon* criaram o *Logic Theorist* (LT), um programa que permitia resolver problemas de forma lógica, conseguindo mesmo provar alguns teoremas sozinho, ao encontrar expressões verdadeiras, fazendo uso de heurísticas, até alcançar

um axioma. Criaram também o *General Problem Solver* (GPS), o qual imitava o processo humano de resolver problemas. Outros programas idênticos foram construídos, fazendo normalmente fazendo uso de informações externas para procurar soluções válidas para o problema em causa.

Durante 1962 foram feitos vários avanços na área das redes neuronais, como por exemplo o Perceptrão. Este tinha a capacidade de incluir mais do que uma camada de neurónios, o que permitia aprender padrões mais complexos [15]. Nos anos 80 generalizou-se o uso do algoritmo de retro-propagação do erro, o qual possibilitava o ajuste automático dos pesos nas ligações das redes neuronais. A utilização destas passou a denominar-se de abordagem conexionista, distinguindo-se da abordagem mais computacional utilizada no início da inteligência artificial, onde regras eram fornecidas e procuras eram feitas para se encontrar uma solução.

Em 1958, uma outra abordagem, referida como *Machine Evolution* em [16] e [6], também foi explorada e posteriormente referida como algoritmos genéticos em [18]. Esta utiliza operações genéticas como mutação, recombinação e seleção natural para procurar a melhor solução em espaços de procura extensos e complexos.

Desta forma, é possível constatar que agentes racionais podem ser implementados usando diferentes abordagens, desde procurar soluções de forma automática com base em regras previamente definidas, até ajustar parâmetros entre ligações ou aplicar operações genéticas de forma a alterar e avaliar estruturas de dados como possíveis soluções, destacando assim a abordagem racional, não baseada em lógica pura, mas sim numa aproximação à melhor solução possível, tendo em conta as entradas fornecidas.

## 2.1. *Deep Learning*

Como foi constatado no capítulo anterior, as redes neuronais foram e são uma metodologia importante, incluindo várias abordagens diferentes, cada uma com um intuito e conjunto de problemas específico. O tipo mais simples e conhecido são as redes neuronais diretas profundas (*Deep Feedforward Networks*) que ao contrário de um perceptrão isolado, que tem apenas um neurónio, são constituídas por múltiplos neurónios em diferentes camadas, denominando-se também de perceptrão multicamada (*MultiLayer Perceptron*), permitindo modelar funções complexas que transformam um conjunto de entrada num conjunto de saída como referido em [19].

De forma mais detalhada, pode-se pensar num perceptrão como sendo uma transformação do *input* num *output*:

$$f(x) = y \qquad \mathbf{1}$$

À estrutura onde ocorre a transformação chama-se neurónio, devido à similaridade, que foi mostrada anteriormente, com um neurónio do cérebro humano. No entanto, num MLP (*MultiLayer Perceptron*) não há apenas uma transformação, há múltiplas

transformações consecutivas correspondendo a diversas camadas de neurónios alimentadas pelas camadas anteriores, como por exemplo:

$$k(g(f(x)) = y \quad \textbf{2}$$

Assim, o resultado de $f(x)$ é a entrada da função $g$ e o resultado desta é a entrada de $k$, obtendo assim o resultado, $y$. Constata-se assim que uma rede neuronal é uma função composta por outras funções compostas:

$$(k \circ (g \circ f))(x) \quad \textbf{3}$$

Assim, uma rede neuronal é um conjunto de camadas escondidas, uma para cada função de transformação, onde cada camada tem um número de unidades. Estas unidades calculam valores, resultantes da transformação entre os pesos associados a cada entrada, permitindo que haja uma transformação entre os mesmos inputs, $x$, com diferentes pesos. A cada uma destas unidades denominamos de neurónio.

Desta forma, cada camada pode ser representada por uma matriz de pesos que é multiplicada pela matriz de entradas, $x$, sendo esta multiplicação a função que transforma as entradas, $x$. Porém, devido à complexidade do problema, transformações lineares nem sempre conseguirão modelar uma relação entre $x$ e $y$. Por isso após a transformação linear é necessária uma transformação extra, sendo esta uma função não linear, denominada de função de ativação, havendo uma grande diversidade destas funções. Assim cada unidade de uma camada terá como saída o resultado da transformação não linear e no fim minimiza-se uma função de custo que considera os erros destas transformações. A isto denomina-se de *Adalines* [20], pois o perceptrão minimiza o erro das estimativas, já as *Adalines* minimizam uma função de custo, devido à utilização de funções de ativação contínuas.

A última camada é onde o resultado, $y$, é calculado, sendo a função de ativação diferente das demais, e normalmente é denominada como camada de saída.

Na Figura 2, tem-se a representação gráfica de uma rede neuronal sob a forma de um grafo acíclico direcionado, conhecido como DAG (Directed Acyclic Graph), isto porque a rede é direcionada para frente, ou seja, o resultado da transformação é sempre passado para uma unidade da próxima camada, não havendo ligações para trás ou para unidades da mesma camada. Trata-se de uma rede neuronal 4x2x1, pois tem 4 entradas, uma camada escondida com 2 unidades e uma camada de saída com 1 unidade. A profundidade da rede corresponde ao número de camadas escondidas e de saída, neste caso é 2. A largura da rede é medida pelo número de unidades, ou seja, pelo número de elementos no vetor de cada camada escondida.

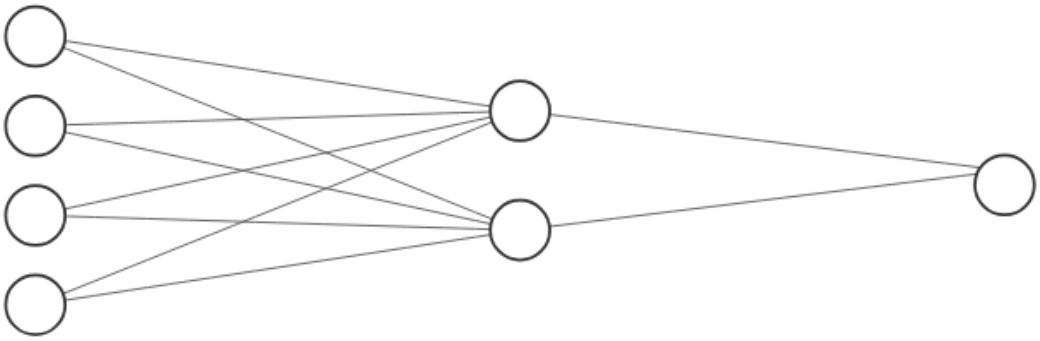

Figura 2 - Rede neuronal 4x2x1

Depois de os valores de ativação dos neurónios serem calculados, sequencialmente desde as entradas até à camada de saída, é necessário calcular o erro nesta camada e propagar o mesmo através de cada peso para as unidades das camadas escondidas. Assim cada peso terá um determinado impacto no valor do erro do neurónio de onde parte a respetiva ligação. Este processo é implementado através do algoritmo de retro-propagação do erro [20], sendo que este faz uso da regra da derivação em cadeia, podendo-se decompor uma função em várias funções compostas e calcular as derivadas relativas a certos parâmetros até se alcançar o parâmetro que se deseja. Para o exemplo seguinte:

$$\big(k \circ (g \circ f)\big)(x) \qquad \mathbf{4}$$

onde $k$ seria a função de ativação da camada de saída, logo na retro-propagação pretende-se encontrar: $\frac{\partial k}{\partial x}$. Porém $k$ não depende de $x$, mas sim da função $g$ e esta depende de $f$ que posteriormente depende de $x$, havendo assim, uma relação na variação de cada uma das funções, pelo que segundo a regra da cadeia:

$$\frac{\partial k}{\partial x} = \frac{\partial k}{\partial g} \cdot \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x} \qquad \mathbf{5}$$

Desta forma é possível obter-se o erro entre $x$ e $y$, além de todos os erros entre cada função intermédia. Como cada função é uma matriz de pesos transformada, não linearmente, por uma função, $\phi$, então é possivelmente calcular o gradiente (matriz de derivadas parciais) para cada função e atualizar os seus pesos. Assim cada matriz de pesos em cada camada será atualizada de acordo com:

$$w := w + \triangle w \qquad \mathbf{6}$$

sendo $\triangle w$ a derivada parcial e $w$ os pesos atuais, [21]. Normalmente, a derivada parcial nunca é aplicada na totalidade, permitindo que o algoritmo ajuste mais rápida

ou lentamente os pesos. Para controlar a rapidez de aprendizagem um fator denominado de ritmo de aprendizagem ou *learning rate*, $\eta$, é multiplicado à derivada parcial:

$$\triangle w := -\eta \frac{\partial \phi}{\partial w} \quad \textbf{7}$$

Sendo $\partial\phi$ a derivada parcial da função de ativação da camada, $\partial w$ os pesos que a derivada parcial considerará e $\eta$ é o ritmo de aprendizagem. Coloca-se o sinal negativo, pois normalmente pretende-se minimizar uma função de custo, como o erro, logo o gradiente deve ser descendente e não ascendente como ele é naturalmente. Como estas redes tendem a ter uma quantidade significativa de pesos, utiliza-se um gradiente descendente estocástico [21], de forma a minimizar o custo computacional do processo de otimização tendo também a vantagem de ajudar na generalização da aprendizagem. Estocástico significa que este não utilizará todas as entradas do conjunto de entradas, selecionando apenas algumas de forma aleatória.

Por fim, é importante destacar as mais comuns funções de ativação, começando pelas funções de custo que são aplicadas na camada de saída. Nesta pretende-se calcular a seguinte probabilidade [19]:

$$p(y\,|x;\,\theta) \quad \textbf{8}$$

com $y$ sendo o resultado empírico das entradas, $x$ as entradas e $\theta$ todos os pesos da rede, ou seja, tendo as entradas e os pesos como se pode obter os resultados pretendidos. Medidas como média dos erros quadráticos (MSE) não dão bons resultados em redes neuronais profundas, utilizando-se antes a entropia cruzada, pois permite verificar o quanto a distribuição estimada pela rede é próxima da distribuição empírica. Isto deve-se ao facto de que esta, a entropia cruzada, é uma função com um gradiente grande e bem definido, o que mantém também as derivadas bem definidas, ao contrário de outras funções que são demasiado retas, resultando em derivadas quase nulas, o que prejudica a aprendizagem da rede. No entanto, a entropia cruzada não é uma função de ativação, ela apenas calcula a aproximação de duas distribuições (função de custo ou de erro), $p$ e $q$, sendo $p$ a distribuição de probabilidade empírica. Logo, ainda é necessária uma função de ativação que transforma as entradas da penúltima camada para a última.

Para problemas de classificação binária é comum utilizar a função de ativação sigmoide, $\sigma$:

$$\sigma(z) = \frac{1}{1 - e^{-z}} \quad \textbf{9}$$

com $z$ sendo a saída da penúltima camada. A sigmoide tem problemas com o desaparecimento do gradiente, devido ao seu intervalo $[0,1] \in \mathbb{R}$, ou seja, gradientes a tender para $-\infty$, ficam 0, e gradiente a tender para $+\infty$, ficam 1. Para classificação de múltiplas classes é comum utilizar-se a função *softmax*, onde a probabilidade de a entrada, $x$, pertencer a uma classe $y_i$, é calculada para cada classe, $i$, da seguinte forma:

$$softmax(z_i) = \frac{e^{-z_i}}{\Sigma_j e^{-z_j}} \quad \textbf{10}$$

Para camadas escondidas há 3 tipos de função de ativação comuns, Figura 3. A primeira delas denomina-se ReLU (*Rectified Linear Unit*):

$$ReLU(z) = \max(0, z) \quad \textbf{11}$$

Esta função mantém os gradientes grandes, já que atua como uma função identidade para os valores maiores que 0. A desvantagem é que se houver muitos valores menores ou iguais a 0 o gradiente será nulo, não havendo assim qualquer aprendizagem. Algumas alternativas foram criadas, como a *leaky* ReLU [22], de forma a manter valores negativos. Outras possibilidades são função sigmoide, que fora apresentada anteriormente, e a tangente hiperbólica:

$$\tanh(z) = 2\sigma \cdot 2z \quad \textbf{12}$$

onde, $\sigma$ é a sigmoide de z. Tanto a sigmoide como a tangente hiperbólica têm o problema de desaparecimento de gradiente, porém a $tanh$ tem um contradomínio $[-1, 1] \in \mathbb{R}$, permitindo assim gradiente de números negativos. Normalmente o uso de funções que saturam, como estas duas é desencorajado.

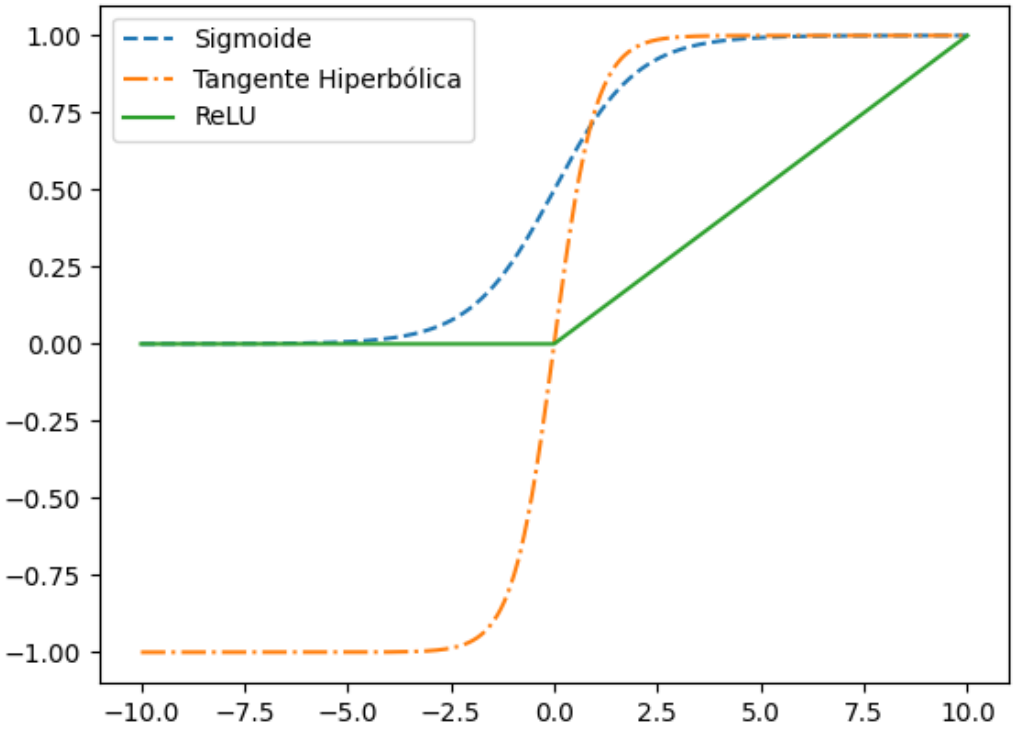


Figura 3 - Funções de ativação

Por fim, considerando que os problemas analisados neste projeto são todos de classificação, é importante mostrar algumas métricas de avaliação. Em [23] são apresentadas diversas métricas, porém para simplificar as suas definições, introduzir-se-á a matriz de confusão, Tabela 1, que permite verificar a frequência de classificações corretas e erradas. Assim tem-se quatro células, cada um representa a frequência dos seguintes tipos de classificações:

- TP (*True Positive*): Empiricamente positivo e classificado como positivo
- FP (*False Positive*): Empiricamente negativo, mas classificado como positivo. Denomina-se erro do tipo I
- TN (*True Negative*): Empiricamente negativo e classificado como negativo
- FN (*False Negative*): Empiricamente positivo, mas classificado como negativo. Denomina-se erro do tipo II

Esta tabela pode ser estendida para classificações não binárias, porém para manter a simplicidade, manter-se-á apenas este tipo de matriz de confusão.

Tabela 1 - Matriz de confusão

| | | Classes estimadas | |
|---|---|---|---|
| | | Positivo (1) | Negativo (0) |
| Classes reais | Positivo (1) | TP | FP |
| | Negativo (0) | FN | TN |

A exatidão (*accuracy*) é a taxa de acertos:

$$acc = \frac{TP + TN}{TP + TN + FP + FN} \quad \textbf{13}$$

A taxa de erro é o complemento da exatidão:

$$err = 1 - acc \quad \textbf{14}$$

Outra medida importante é a precisão, já que esta mede o que fora classificado como positivo, sendo sensível aos erros do tipo I:

$$prec = \frac{TP}{TP + FP} \quad \textbf{15}$$

A sensibilidade (*recall*) ou taxa de verdadeiros positivos (TPR), mede o rácio entre o que foi classificado como positivo e o que realmente era positivo. Esta é sensível a erros do tipo II. Assim, quanto mais o valor se aproxima de 1.0, melhor:

$$recall = \frac{TP}{TP + FN} \quad \textbf{16}$$

A medida que relaciona estas duas últimas, tendo assim um nível de robustez superior à exatidão, é o F-*score*:

$$FScore = \frac{2 \times prec \times recall}{prec \times recall} \quad \textbf{17}$$

Uma medida gráfica é o ROC (*Receiver Operating Characteristics*), este permite relacionar a sensibilidade com a taxa de falsos positivos (FPR):

$$FPR = \frac{FP}{FP + TN} \quad \textbf{18}$$

Assim quanto maior a sensibilidade e menor o FPR, como ilustrado na Figura 4. Esta permite analisar as classificações positivas de múltiplos classificadores, sendo que os melhores classificadores terão mais curvas à esquerda do gráfico.

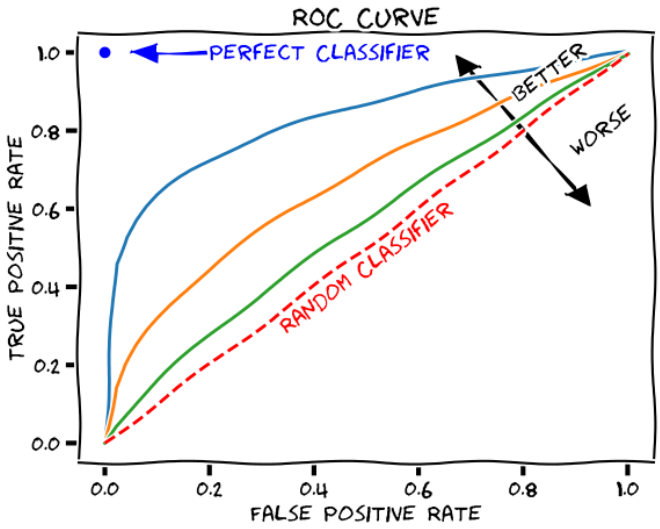


Figura 4 - Curva ROC

Para reduzir a curva ROC a um número, número este que deve mostrar o quão boa a curva é, calcula-se a área por debaixo da curva. Este valor é denominado de ROC AUC *score* (ROC *Area Under Curve score*).

Ao entender-se redes neuronais profundas e todas as funções e métricas comumente utilizadas, considerando principalmente problemas de classificação, introduzir-se-á um novo tipo de rede neuronal. Esta é mais apropriada para o tipo de dados (imagens) com que se trabalhará neste projeto.

### 2.1.1. Redes Neuronais Convolucionais

Redes neuronais convolucionais foram apresentadas em [1], sendo a convolução uma operação matemática que permite fazer uma soma ponderada de partes de um vetor ou matriz de entrada:

$$conv(x) = (M * w)(x) \quad \textbf{19}$$

com $M$ sendo o vetor/matriz de entrada, $w$ é o conjunto de pesos, denominado de *kernel* ou núcleo, e $x$ é a porção da matriz onde será aplicada a convolução. É importante que $w$ seja uma função de densidade de probabilidade, caso contrário não há uma soma ponderada como referido em [19]. Para definir a operação de convolução em dados discretos, como a representação de pixéis numa imagem, utiliza-se a seguinte fórmula:

$$conv(i,j) = (w * M)(i,j) = \Sigma_m \Sigma_n M(i-m, j-n) \cdot w(m,n) \quad \textbf{20}$$

Nesta fórmula é necessário inverter o *kernel*, movendo-o em relação à entrada, o que para a operação de convolução é importante, principalmente para o rigor matemático. Contudo, em *Deep Learning*, não há essa necessidade, porque a própria rede irá aprender os pesos, utilizando-se assim uma correlação cruzada:

$$conv(i,j) = (w * M)(i,j) = \Sigma_m \Sigma_n M(i+m, j+n) \cdot w(m,n) \quad \textbf{21}$$

As motivações para utilizar convolução, descritas em [19], relacionam-se com as interações esparsas. Isto é, nas redes neuronais diretas profundas, cada entrada interage com todas as unidades. No entanto, como as entradas numa imagem são pixéis, analisar pixel a pixel não será algo útil, pois os contornos duma imagem ou outros atributos da mesma ocuparão pequenas regiões de pixéis. Utilizando a convolução poupa-se no tempo de execução ao trabalhar-se em pequenas regiões e não entrada a entrada. Outra motivação consiste, na partilha de pesos, já que o *kernel* tem um tamanho fixo e os pesos também, logo em vez de se aprender um peso para cada conexão, aprende-se um pequeno conjunto deles que será utilizado em todas as sub-regiões da imagem. Minimiza assim o espaço necessário para armazenar pesos, já que há significativamente menos do que numa rede neuronal direta profunda. Outra vantagem desta partilha é a necessidade de aprender pesos que funcionem com diferentes pixéis, possibilitando aprender relações entre várias variáveis. Por fim, esta motivação leva a outra, sendo esta a equivariância. Isto significa que, se uma imagem for transformada por uma função $f$ e por uma outra função $g$, não importa ordem com que se aplique cada uma delas, o resultado será sempre o mesmo. Logo, se um objeto alterar a sua posição, esta também será alterada na saída, tornando-se uma vantagem no processamento de múltiplas imagens com um mesmo objeto.

Normalmente, convoluções são aplicadas em imagens com múltiplos canais, ou seja, o *kernel* não é uma matriz e sim um *tensor* 3D (comprimento, largura, profundidade). Utilizam-se vários *kernels* diferentes, de forma a retornar múltiplas saídas. Além disso, convoluções diminuem as dimensões espaciais da imagem (comprimento e largura). Quanto maior o tamanho do *kernel*, maior a diminuição, quando se usa *valid padding*, ou seja, a imagem não sofre qualquer tipo de padding. Como diminuir a dimensão da imagem significa perder dados, por vezes não é a melhor opção, por isso existe o *same padding*, onde são adicionados pixéis com valor 0 ao redor da imagem, de forma que após a convolução, o tamanho espacial permaneça. Por fim, o *full padding* coloca pixéis

com valor 0 à volta da imagem, garantindo que o kernel passa por todos os pixéis o mesmo número de vezes, porém este é menos utilizado, devido a fazer com que a rede aprenda menos, pois os pixéis nas bordas tendem a ter menos importância que os centrais, logo manter a importância igual para todos nem sempre é a melhor opção.

O último parâmetro importante de uma convolução é a *stride*, sendo que esta define o quanto o *kernel* se desloca a cada convolução. Uma convolução ocorre, quando esta é aplicada apenas a uma região da imagem, resultando num novo pixel que será colocado, na mesma localização, na imagem de saída. Logo uma imagem completa sofrerá múltiplas convoluções. Para saber qual será a próxima região onde a convolução é aplicada, a *stride* indicará qual será o deslocamento nos diferentes eixos da imagem. Logo, quanto maior a *stride* menor será o tamanho da saída.

Uma camada convolucional começa por fazer múltiplas convoluções na imagem, com múltiplos *kernels* distintos, resultando em múltiplas saídas distintas. A Figura 5 é o exemplo de uma camada convolucional com *valid padding*, 1 *stride*, 1 *kernel* e 1 saída. Se fossem necessárias 32 saídas, seriam necessários 32 *kernels*.

Estas convoluções são o primeiro estágio de uma camada convolucional. No entanto, há mais duas etapas a realizar [19].

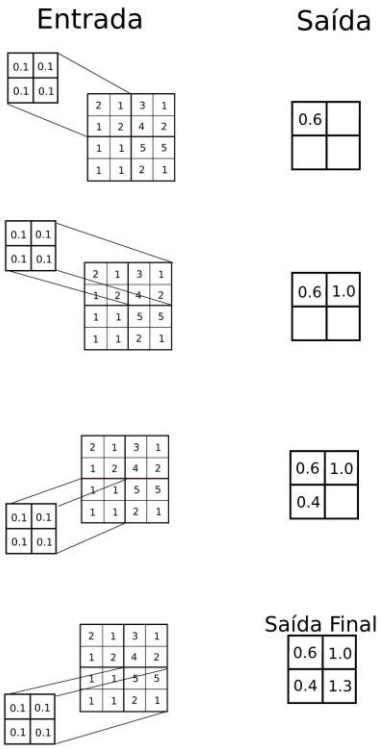


Figura 5 - Exemplo de funcionamento de uma convolução

A próxima etapa consiste na aplicação de uma função de ativação a todas as saídas geradas pelo estágio anterior. A função mais comum é a ReLU.

A última etapa, corresponde à utilização de uma camada de *pooling*, onde é executada a operação de *pooling* à entrada. Esta operação é idêntica à de convolução no sentido em que também tem *kernel*, *stride* e *padding*, não havendo, porém, pesos no *kernel*. Este *kernel* funciona como uma janela deslizante e dependendo do tipo de operação de *pooling*, obterá valores diferentes, gerando no fim o mesmo número de saídas que a entrada, mas reduzindo o tamanho espacial da imagem de entrada.

Se a operação de *pooling* for do tipo *average* (média), a mesma fará a média dos valores dos pixéis na região onde a janela deslizante se encontra e coloca o valor como um pixel da saída na mesma posição. Se for do tipo *maximum* (máximo), colocará o maior valor de pixel na região.

Este tipo de operação cria invariância a pequenas translações, sendo diferente da equivariância das convoluções, pois esta invariância é criada através de pequenas mudanças de posições. Por exemplo, os olhos de uma pessoa não precisam estar perfeitamente alinhados para se entender que uma face é de uma pessoa, estes devem estar próximos um do outro, mas não exatamente sempre à mesma distância.

Ao gerar múltiplas saídas no estágio convolucional, pode-se utilizar as *poolings* de forma a estas aprenderem a que tipo de transformações se devem tornar invariantes.

A camada de *pooling* também serve como uma etapa de *downsampling*, ou seja, como forma de diminuir o tamanho espacial da entrada.

## 2.2. Algoritmos Evolucionários

Como mencionado em [24], em computação evolucionária não se ajustam pesos entre conexões, mas segue-se antes o mecanismo de seleção natural como ocorre na natureza: Nesta há variações nos indivíduos que nela habitam, resultantes de mutações e recombinação que ocorrem ao nível do material genético. Apenas os indivíduos que melhor se adaptam ao ambiente sobrevivem e produzem filhos que passam para a próxima geração. Simulando este processo computacionalmente espera-se obter soluções qualidade crescente para problemas complexos e com um espaço de procura muito extenso. Pode-se considerar uma técnica de otimização meta-heurística, pois não há uma heurística geral, permitindo antes a criação de uma heurística específica para cada problema.

Estes algoritmos baseiam-se nas características reais da biologia, porém de forma simplificada. Todos os seres são compostos por células que têm, cada uma, cromossomas, sendo estes divididos em genes que codificam uma proteína específica. A posição de cada gene denomina-se de *locus*.

A coleção de todos os cromossomas denomina-se de genoma e o conjunto específico de genes que constitui um genoma é denominado de genótipo. A um nível mais abstrato e após desenvolvimento, o genoma cria o fenótipo que possui as características físicas e mentais do indivíduo.

Os cromossomas podem ser diploides, arranjados em pares, ou haploides sem qualquer tipo de arranjo. Na recombinação, ou seja, na reprodução sexual, seres com cromossomas diploides, trocam genes entre pares de cromossomas, gerando novos cromossomas diploides. Já os haploides, partilham genes entre si, gerando cromossomas simples. A mutação ocorre nos nucleotídeos do filho, comumente denominado de *offspring*. O *fitness* ou desempenho de um indivíduo constitui uma medida da probabilidade de o mesmo sobreviver e se reproduzir.

Em algoritmos genéticos clássicos, cromossomas serão as soluções candidatas, codificados, normalmente, numa string binária. Os genes são os bits dentro da string ou blocos de bits. Os alelos são os valores possíveis para cada bit (0 ou 1): Caso a representação não seja binária, haverá mais alelos. A recombinação é feita como nos cromossomas haploides e as soluções têm apenas um cromossoma. A mutação é feita invertendo um bit em um locus aleatório. Neste tipo de algoritmos não é comum haver fenótipo.

A forma de avaliar se uma solução/indivíduo passa para a próxima geração, depende do problema, no caso deste projeto, pretende-se otimizar uma rede neuronal, logo a função de avaliação poderia ser uma medida do desempenho da rede, como a acurácia, caso se pretenda um problema de maximização, ou a taxa de erro, no caso de um problema de minimização. Este processo está ilustrado na Figura 6.

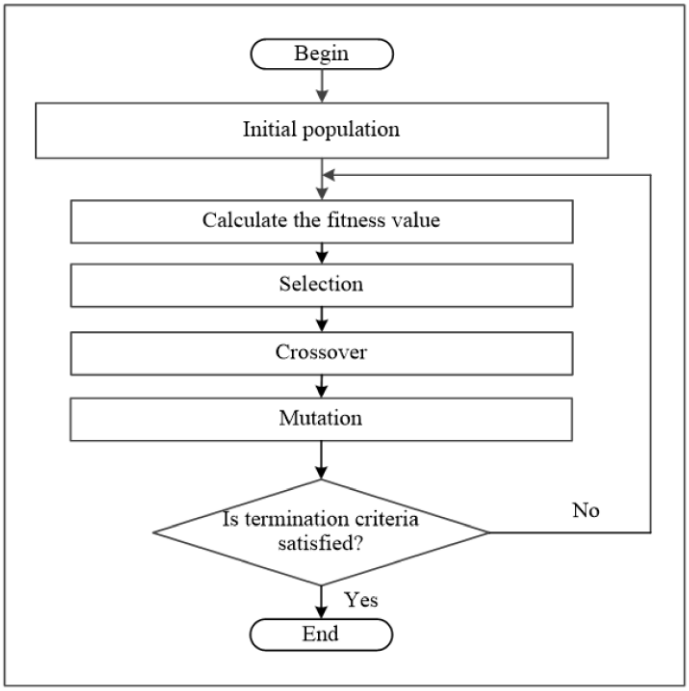


Figura 6 - Etapas do algoritmo evolucionário [25]

Como mostrado em [26], há múltiplas formas de selecionar soluções para prosseguirem para a próxima geração, assim como mutações e recombinações.

Em termos de seleção a mais comum é a seleção por torneio, onde um conjunto de soluções são selecionados através com base no *fitness* utilizando uma roleta, onde

quem tem maior *fitness* tem maior probabilidade de ser selecionado. O indivíduo com maior *fitness* é colocado na *pool* de indivíduos para a próxima geração.

O elitismo garante que um certo número de melhores indivíduos tem passagem garantida para a próxima geração, evitando assim que os melhores indivíduos se percam aquando da aplicação da recombinação e mutação

Em operadores de mutação também há algumas opções como a mutação por deslocamento, onde um pequeno conjunto de genes é movido para uma posição aleatória do genótipo. Na mutação por inversão, troca-se a posição entre 2 genes e a mutação por embaralhamento, embaralha os genes numa região aleatória do genótipo. A Figura 7 mostra visualmente como estas mutações funcionam.


Figura 7 - Exemplos de mutações

Os operadores de recombinação são vários, como a recombinação de um ponto. Nesta, o ponto é selecionado aleatoriamente, em ambos os indivíduos, e informações os segmentos após desse ponto são trocadas entre os dois progenitores, gerando dois novos filhos. Há uma generalização deste tipo de recombinação em que há k pontos, e os genes entre esses k pontos seriam trocados entre os indivíduos. A recombinação uniforme vai de gene em gene, decidindo aleatoriamente se há troca com o gene correspondente do outro indivíduo ou não.  A Figura 8 apresenta as recombinações visualmente.


Figura 8 - Exemplo de recombinações

São estes operadores genéticos que possibilitam a evolução dos indivíduos ao longo de várias gerações, garantindo que exploram uma grande diversidade de combinações, mantendo-se numa direção propensa a valores *fitness* melhores.

## 2.3. Inteligência de Enxame

Um exame é um vasto conjunto de agentes simples que interagem entre si para alcançar um objetivo comum. As interações podem ser diretas, como contacto ocular ou auditivo, ou indiretas, como a alteração do ambiente por parte de um indivíduo que obrigará os demais a adaptarem-se, como é explicado em [27]. Aqui também se pode considerar que esta é técnica de otimização meta heurística.

A primeira aplicação computacional deste tipo de inteligência foi introduzida no ano de 1989, em [28], e em 1991 foi apresentado o algoritmo de otimização da colónia de formigas (ACO) [29]. Este baseia-se no comportamento das formigas em colónias, já que estas conseguem encontrar o caminho mais curto entre a comida, e o seu ninho, Figura 9, sem qualquer uso da visão. Elas comunicam através de feromonas, ao libertarem-nas no caminho de forma a guiarem-se umas às outras.

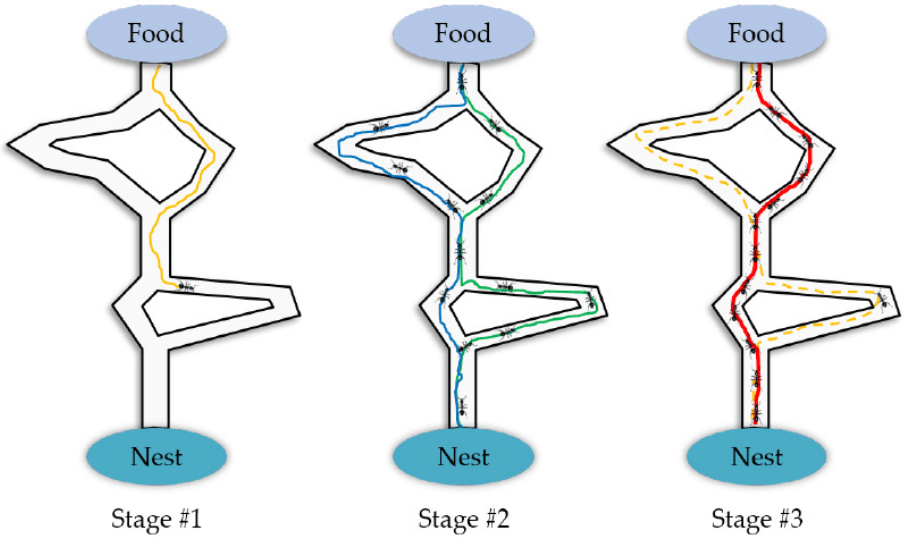


Figura 9 - Convergência para o caminho mais curto na ACO [30]

Outro famoso algoritmo foi criado em 1995 [31], denominando-se *Particle Swarm Optimization*. Baseia-se no comportamento de bandos de pássaros que se organizam para encontrar comida, sem colidirem e reagrupando-se de forma rápida. Sendo este um algoritmo muito utilizado, será explicado com maior detalhe.

Como é descrito em [27], os pássaros podem ser considerados partículas que partilham uma direção para se moverem, as quais consideram os vizinhos mais próximos para ajustar velocidade e a posição. Devem também evitar colisões garantindo assim a distância mínima entre partículas e mantendo uma certa velocidade. O PSO resolve um problema utilizando um conjunto (enxame) de pontos definidos num espaço n-dimensional, onde a velocidade de cada ponto é dinâmica e ajustável. Todas as partículas sabem qual foi a sua melhor posição já avaliada (*fitness*) bem como a melhor posição encontrada até agora pelo enxame e utilizam essa informação para ajustar a sua posição a cada iteração. Assim, o bando aproxima-se do ponto mais promissor do espaço, considerando o seu histórico, bem como o do enxame, Figura 10.

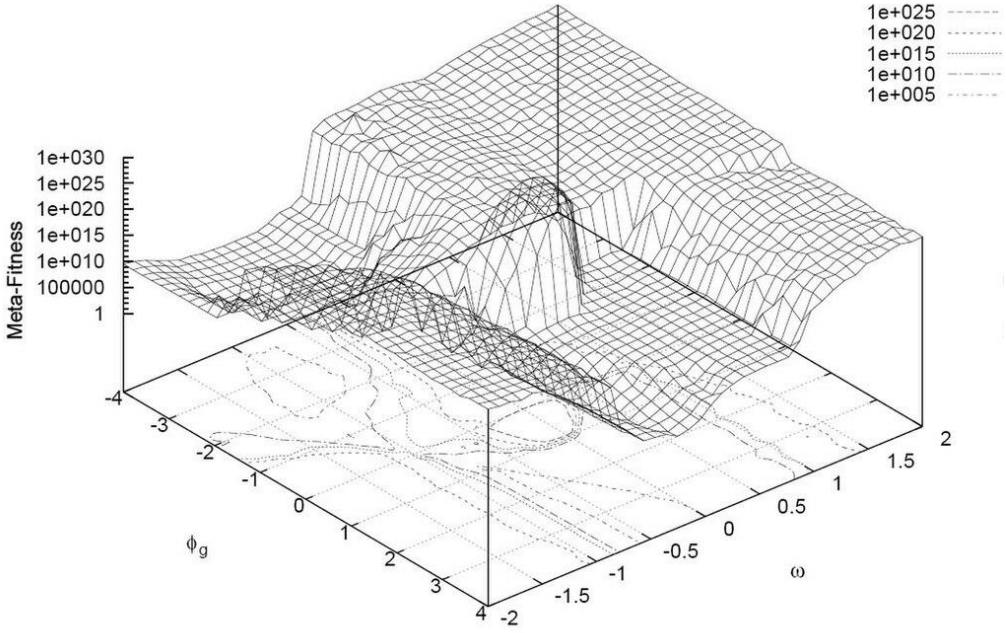


Figura 10 - Convergência para a posição ótima no PSO

Além da sua melhor posição, as partículas sabem a partícula com melhor *fitness*, ajustando a sua velocidade face a essa partícula ou utiliza na mesma o seu próprio histórico para se ajustar.

Para atualizar-se as velocidades, utiliza-se a fórmula:

$$v_i^d(t+1) = v_i^d(t) + c_1 \cdot R_1 \left(p_i^d(t) - x_i^d(t)\right) + c_2 \cdot R_2 \left(p_{gd}(t) - x_i^d(t)\right) \qquad \mathbf{22}$$

Onde $v_i^d$ é a velocidade da $i$-ésima partícula na dimensão $d$, $c_1$ e $c_2$ são pesos fixos que definem a importância da componente própria ou do enxame, $R_1$ e $R_2$ são valores contínuos aleatórios uniformes $\in [0,1]$ permitindo aleatoriedade na atualização, $x_i^d$ é a posição da $i$-ésima partícula na dimensão $d$, $p_i^d$ é a melhor posição da $i$-ésima partícula na dimensão $d$ e $p_{gd}$ é a posição da melhor partícula do bando.

A utilização da velocidade atual na fórmula faz com que esta se comporte como inércia, mantendo a direção sem mudanças drásticas. O termo seguinte causa uma atração ao ponto da melhor posição da própria partícula e o final causa uma atração à posição da melhor partícula.

Para atualizar-se as posições, utiliza-se:

$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1) \qquad \mathbf{23}$$

Aqui utiliza-se a nova velocidade calculada para ajustar a posição da partícula.

O algoritmo tem semelhanças com o algoritmo genético, só que não há operações genéticas e é necessário guardar o histórico de cada partícula. O método de avaliação (*fitness*) também depende do problema.

Esta técnica tem uma rápida convergência, beneficiando de resultados anteriores para se adaptar, logo tendo um potencial para se adaptar a modificações no ambiente. No entanto, tem limitações quanto à sua representação vetorial que nem sempre é possível, além de que assume que todas as partículas são homogéneas, comportando-se todas de igual maneira.

# 3. Estado da Arte

O objetivo do projeto é estudar e avaliar as diferentes metodologias para se encontrar arquiteturas de redes neuronais convolucionais, utilizando algoritmos evolucionários e inteligência de enxame, garantindo que se possam executar em hardware limitado e tempo reduzido.

Para se obter o estado da arte de algoritmos com estas características, optou-se por utilizar a estratégia de revisão sistemática [32]. Neste tipo de revisão, Figura 11, primeiramente, planeia-se o assunto e as perguntas que pretendem ser respondidas com a revisão, depois procura-se artigos nas diversas bases de dados, utilizando uma expressão de procura. De seguida, filtra-se os artigos considerando alguns critérios de exclusão e inclusão e por fim analisa-se os artigos e escreve-se a revisão.

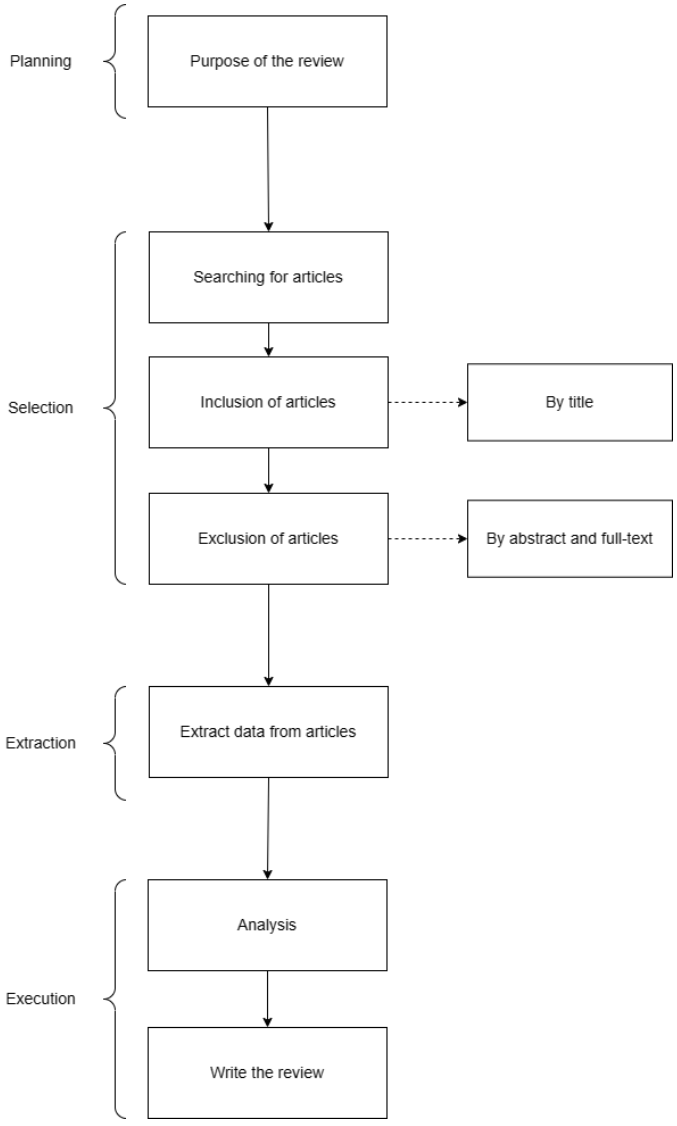


Figura 11 - Fluxograma da revisão sistemática

Para auxiliar na gestão de todos os artigos selecionados utilizou-se a ferramenta *Mendeley* [33]. Esta permite gerir artigos científicos, inclusive importá-los através de ficheiros exportados de bases de dados (como ficheiros .ris e .bib). É possível instalar

extensões para o navegador, de forma, a adicionar automaticamente as referências dos artigos lidos à conta do *Mendeley*. Caso se instale o plugin para o Word, é ainda possível, criar as referências bibliográficas automaticamente, com o estilo de citação desejado.

A primeira fase da revisão está explicitada no primeiro capítulo do relatório. Para procurar os artigos utilizaram-se as bases de dados: ACM[34], IEEE[35] e Scopus [36] – fazendo uso da seguinte fórmula de procura:

**(“convolutional neural network”) AND ("neuroevolution" OR "evolutionary algorithm" OR "evolutionary computing" OR "neural architecture search")**

Foi possível obter 1621 artigos, com 177 duplicados, sendo a procura executada no dia 26 de setembro de 2023. Os artigos foram, primeiramente, selecionados por título, considerando os seguintes critérios de inclusão: mencionar redes neuronais convolucionais, enfatizar evolução ou procura de arquiteturas de redes neuronais e mencionar classificação de imagens, apesar deste último critério ser opcional. Desta forma, foi possível diminuir de 1475 para 146 artigos.

Após leitura dos *abstracts* e a utilização dos seguintes critérios de exclusão: não indicar que pretendem minimizar o tempo ou não indicar que utilizam apenas 1 GPU, uso de *datasets* específicos ou pouco conhecidos, artigos escritos em outra língua diferente de inglês – permitiu reduzir o número de artigos para 21, porém após leitura completa, outros 7 artigos foram excluídos, finalizando com um total de 14 artigos.

No próximo subcapítulo serão analisados cada um dos artigos para que no fim seja possível analisar, comparar e concluir quais são as melhores metodologias.

## 3.1. Artigos selecionados

Aqui serão analisados e resumidos os artigos selecionados, alguns destes, após análise, mostraram que utilizavam mais que uma GPU, porém se a metodologia for diferente e apresentar bons resultados, não será descartada e a análise será feita de qualquer forma. A Tabela 2 contém informação acerca dos hiperparâmetros das redes convolucionais durante a execução do algoritmo evolucionário, a Tabela 3 contém os parâmetros do algoritmo evolucionário e a Tabela 4 contém informações sobre o hardware utilizado.

### Lamarckian Evolution of Convolutional Neural Networks

Este método [37] utiliza blocos que contêm: uma convolução, uma *batch normalization* (camada que normaliza os valores de uma matriz) e uma função de ativação ReLU. Não é referido o tipo de representação do genótipo, porém presume-se ser uma simples lista de blocos.

Cada bloco tem os seguintes parâmetros para se otimizarem: Número de *kernels*, tamanho do *kernel* e tamanho da *stride*.

O próprio indivíduo contém o parâmetro: número de blocos na rede neuronal – para ser otimizado.

A estratégia utilizada é denominada de (1+1), ou seja, um indivíduo pai, inicialmente é apenas uma rede com um bloco, gera um filho após a aplicação dos operadores genéticos. Neste caso, apenas a mutação é aplicada. O filho é avaliado por um número estático de épocas (número treinos feitos com um *dataset*), no *dataset* de validação, sendo a exatidão de validação o seu valor de *fitness*. Este processo faz uso da retro-propagação para treinar o indivíduo.

Se o *fitness* do filho for superior ao do pai, então ele substitui o pai. No entanto, para evitar que o algoritmo fique preso em mínimos locais (já que o algoritmo é ganancioso), há uma probabilidade, $\eta$, do filho, mesmo tendo pior *fitness,* ser selecionado para uma chamada recursiva, onde o mesmo será o pai, repetindo esse processo $k$ vezes. O indíviduo com melhor *fitness,* nessa chamada recursiva, é comparado com o indivíduo pai original e se o *fitness* for melhor, substitui-o. Se não, mantém-se, como aconteceria originalmente. Esse conjunto de chamadas recursivas denominam-se de *niching*, já que há uma separação de indivíduos para melhor a diversidade da população.

O operador de mutação tem os seguintes conjuntos de valores possíveis para os parâmetros a otimizar:

- Número de *kernel*: $F = \{16, 32, 64, 128, 192, 256\}$
- Tamanho do *kernel*: $K = \{1, 3, 5\}$
- Tamanho da *stride*: $S = \{1, 2\}$
- Número de blocos $\in \mathbb{N}$

O operador escolhe, aleatoriamente, uma das seguintes mutações, tendo a frequência relativa à frente de cada (quanto maior o número maior a probabilidade):

- 3x Adiciona bloco: Coloca um bloco num qualquer ponto aleatório do genótipo
- 3x Remove bloco: Remove um qualquer bloco aleatório do genótipo
- 2x Adiciona *kernels*: Escolhe uma qualquer convolução e coloca o seu número de *kernels* no valor superior, seguinte ao atual
- 2x Remove *kernels*: Escolhe uma qualquer convolução e coloca o seu número de *kernels* no valor inferior, anterior ao atual
- 2x Mudar tamanho do *kernel*: Escolhe uma qualquer convolução e coloca um valor aleatório retirado de $K$, como tamanho do *kernel*
- 1x Mudar tamanho da *stride*: Escolhe uma qualquer convolução e coloca um valor aleatório retirado de $S$, como tamanho da *stride*

Não pode haver mais que 3 convoluções com *stride* igual a 2, pois estas reduzem as imagens a metade do tamanho e como o *dataset* escolhido tem imagens 32x32, então 3 convoluções com *stride = 2* torná-las-á imagens 4x4.

Os genótipos são guardados, de forma a garantir que as mutações criem sempre genótipos diferentes, ou seja, a mutação é aplicada até um genótipo diferente de qualquer outro anterior, seja gerado.

Para poupar recursos, decidiu-se utilizar a partilha de pesos das redes neuronais convolucionais. Sempre que um novo bloco é adicionado os seus pesos são gerados aleatoriamente, porém após a mutação muitos *kernels* mantêm os valores para os parâmetros, logo esses pesos podem ser preservados. Só se voltam a gerar parâmetros aleatórios caso os parâmetros das convoluções sejam alterados, considerando que se eles forem alterados, o *output* terá um tamanho diferente, logo o tamanho das seguintes convoluções também mudará. Dessa forma elas também terão de gerar novos parâmetros.

Os resultados foram comparados utilizando e não utilizando partilha de pesos, tendo $\eta = 0.1$ e $k = 5$, para a probabilidade de *niching* e o número de repetições, respetivamente. Foram feitas 20 execuções para aliviar a aleatoriedade da procura.

Criaram 2 cenários sem partilha de pesos, um com 4 épocas e outro com 16 épocas para avaliar cada indivíduo ao longo da procura.

Após o algoritmo terminar, treinaram a melhor rede por mais épocas utilizando um *learning rate schedule* (método que ajusta dinamicamente o ritmo de aprendizagem). Utiliza-se o *dataset* de teste para testar a performance desta rede após o re-treino com 512 épocas.

Os resultados com partilha de pesos, no *datatset* CIFAR10, mostraram-se melhores do que os resultados dos 2 cenários, obtendo $85\% \pm 2\%$ contra $82\% \pm 1\%$ do primeiro cenário e $85\% \pm 3\%$ de exatidão, do segundo cenário. Teve um comportamento idêntico no CIFAR100, obtendo $61\% \pm 3\%$ de exatidão. Vale ressaltar que quando treinado com 512 épocas, o cenário 2 leva vantagem de o cenário 1, porque a rede escolhida treinou por mais épocas durante a procura.

Consta-se que quanto maior o número de blocos, melhor a exatidão, com uma média de 7 blocos.

Não houve qualquer comparação de resultados com o estado da arte, porque não havia o objetivo de ultrapassar o estado da arte, mas sim minimizar o tempo de treino, o que foi conseguido devido à partilha de pesos.

### Evolving Image Classification Architectures With Enhanced Particle Swarm Optimization

Neste artigo [38] é utilizado o algoritmo PSO que permite otimizar e treinar arquiteturas de CNNs em simultâneo.

Utilizam, como ponto inicial, uma arquitetura que segue a arquitetura VGG [3], além de utilizarem a técnica de partilha de pesos para otimizarem o tempo de procura.

Cada bloco de camadas é treinado individualmente e um mecanismo de *cosine annealing* é utilizado para balancear a exploração local e global.

Os indivíduos são vetores representados num espaço $n$-dimensional, como é comum neste tipo de algoritmos. Para reduzir o espaço de procura, basearam-se na arquitetura VGG, que explora a diminuição do tamanho espacial e o aumento do número de *feature maps* (saídas de uma camada convolucional). Aqui os blocos seguem o padrão: Convolução, *Batch Normalization*, ReLU e *Max Pooling* – sendo a camada de *pooling* a responsável por diminuir o tamanho espacial.

Todas as convoluções têm um *kernel* 3x3, *stride* 1x1 e *same padding* para manter o tamanho espacial da imagem. Para o número de *kernels* usam: $2^{\epsilon}, \epsilon \in [6,7,8,9]$ para os blocos 0, 1, 2, e 3, respetivamente. Já as camadas de *pooling* reduzem o tamanho espacial para metade, logo têm um *kernel* e *stride* 2x2 com *stride* 2x2.

Para representar estes blocos vectorialmente, os indivíduos utilizam vetores, de inteiros, com tamanho $n$ e com valores no intervalo $[0, 10)$, onde o índice de cada componente do vetor refere-se a um bloco e o inteiro representa o número de camadas a adicionar ao bloco. O *fitness* é o rácio de erro da arquitetura, logo o problema de otimização é de minimização da função de *fitness*, $f(x)$.

Os valores obtidos são do conjunto dos números reais, sendo normalizados para o intervalo estipulado acima e convertidos para inteiros como forma de regularização, sendo isto o que o diferencia do algoritmo PSO original.

O melhor indivíduo local é o melhor indivíduo da população atual, sendo alterado se após a avaliação de um indivíduo da população atual, o melhor tiver *fitness* inferior. Já o global é considerando todas a populações anteriores, sendo alterado de igual forma.

Ao longo das avaliações os pesos são guardados numa *lookup table*, com a chave sendo a combinação do número do bloco e do número de camadas que este tem. O valor será um número referente aos últimos pesos ou aos melhores pesos encontrados até ao momento, para este bloco.

Para garantir que não se limita a otimização dos pesos, criou-se uma fórmula que privilegia os melhores pesos ao longo das gerações. No entanto, usar os últimos pesos mostrou-se a melhor opção. Os parâmetros aprendidos ao longo da procura, são utilizados no treino final da melhor arquitetura encontrada, não sendo necessário re-treinar o modelo de raiz, poupando-se tempo.

Para permitir uma exploração local e posteriormente uma exploração global do espaço de procura, um algoritmo de *cosine annealing* foi aplicado aos coeficientes (pesos) referentes aos melhores indivíduos local e globalmente. A fórmula obtida para este efeito é a seguinte:

$$c_1 = q + \frac{Q-q}{2}\cos\left(\pi\left(1-\frac{t}{T}\right)\right) + 1, \text{com } q = 1.5 \text{ e } Q = 2.5 \qquad \mathbf{24}$$

$$c_2 = q + \frac{Q-q}{2}\cos\left(\pi\left(1-\frac{t}{T}\right)\right) + 1, \text{com } q = 0.5 \text{ e } Q = 2.5 \qquad \mathbf{25}$$

Sendo $t$ a geração atual e $T$ o número total de gerações. Denominaram esta forma de *Cosine Late Crossover*.

No re-treino do modelo, juntaram-se os *datasets* de treino e validação e treinaram-se por mais algumas épocas, no caso 10. Alteraram-se os ritmos de aprendizagem do *cosine annealing schedule*, iniciando com 0.0001 e finalizando com 0.0000001. Este re-treino mostrou-se essencial para reverter o *overffiting* (fenómeno que acontece quando um modelo se ajusta perfeitamente ao conjunto de treino, mas não consegue generalizar para outros dados fora desse conjunto) que o modelo ganhou ao longo da procura.

Os resultados em ambos os *datasets*, CIFAR10 e CIFAR100, mostraram que ter mais camadas nos primeiros blocos leva a modelos mais robustos. Comparativamente, ao estado da arte, este algoritmo mostrou resultados competitivos, considerando o baixo custo computacional e a limitação a nível de recursos, 1 GPU, que o mesmo proporciona.

Este artigo ao utilizar uma partilha de pesos global, evita a duplicação da otimização de parâmetros iguais em múltiplas arquiteturas.

Assim os benefícios gerais da metodologia utilizada neste artigo, são: o treino contínuo dos pesos, preservando o progresso de treino ao longo da execução do algoritmo de procura e não ser necessário um re-treino completo da arquitetura final para assim obter um modelo pronto a utilizar, já que os pesos de vários blocos já estão otimizados, necessitando somente de alguns ajustes para uma melhor generalização. Utilizar o esqueleto de um modelo que mostra resultados satisfatórios nos *datasets* utilizados, além de manter o tamanho espacial dos *kernels* pequenos, 3x3, ajuda tanto a obter modelos precisos e com poucos parâmetros, otimizando também o tempo de treino.

### 3.1.3. *Evolving and Ensembling Deep CNN Architectures for Image Classification*

Este artigo [39] utiliza a metodologia mostrada em [38], porém faz uso de um *ensemble*, ou seja, faz o agrupamento de modelos em que cada um faz a sua classificação e a moda de classificações é selecionada como classificação final. A única mudança foi o valor do parâmetro de inércia que trocou de 0.2 para 0.6.

Para criar o *ensemble* utilizam 2 técnicas: obtêm os $m$ melhores indivíduos que se aproximam da melhor partícula global, denominando a esta técnica de *Local Best Base Model Nomination*. A outra técnica obtém 32 candidatos verificando os melhores pesos para cada bloco e, posteriormente, obtendo os 2 melhores indivíduos que utilizaram

esse bloco, construindo assim 2 novos indivíduos baseado nos melhores pesos por bloco.

Este *ensemble* não sofre de duplicação devido à partilha de pesos que permite uma aprendizagem contínua de cada bloco. Englobando até o treino final em que o modelo selecionado é treinado por mais algumas épocas apenas para evitar *overffiting*.

Os resultados mostraram que a primeira técnica é mais rápida no processo de re-treino, por ter menos indivíduos e que a melhor partícula global tem o melhor resultado. Já na segunda técnica demorou 2 horas para o re-treino e encontraram-se indivíduos com melhores resultados que a melhor partícula global, o que significa que esta técnica permite obter um conjunto de indivíduos mais diversificado. As exatidões mostraram que apesar de demorar mais 2 horas que o [38], no *dataset* CIFAR10, houve uma melhoria de 0.51%. No entanto, para diminuir o tempo, pode-se optar pela primeira técnica, incrementando apenas 15 minutos e obtendo uma melhoria de 0.39%, face ao artigo original.

Este artigo comparativamente ao [38] não introduziu nada de novo para minimizar o tempo e a complexidade do algoritmo evolucionário ou do treino da rede neuronal. No entanto, mostrou melhores resultados com um baixo incremento de tempo, mantendo os recursos.

### A Flexible Variable-length Particle Swarm Optimization Approach to Convolutional Neural Network Architecture Design

Este algoritmo [40] utiliza apenas convoluções com tamanho de *kernel* e *stride* de 3 e 1, respetivamente, além de *pooling* com *stride* e tamanho de *kernel* de 2x2, de forma a minimizar o espaço de procura.

A codificação, para o valor de cada componente do vetor, é binária, utilizando uma *string* de 9-*bits*, sendo que:

- O primeiro *bit* indica se é convolução (0) ou se é *pooling* (1)
- O segundo *bit* é apenas para indicar a mudança de tamanho
- Se for *pooling*, o terceiro bit representa se é *max* ou *average pooling*
- Os restantes *bits* são para a configuração de cada camada, tendo assim a *pooling* valores de [0,127] e a convolução com valores de [256, 384]. Quando os valores não estão dentro dos intervalos, o segundo *bit* é ativo e os valores têm um significado próprio, podendo significar remover, adicionar ou subdividir-se em mais camadas.

Esta representação ultrapassa a limitação do PSO original, onde os indivíduos têm tamanho fixo.

Na geração de indivíduos garante-se que o número de camadas de *pooling* não ultrapassam um certo valor, para evitar erros de construção da rede, já que estas

diminuem o tamanho espacial das camadas a metade. Além disso, de cada vetor tem um tamanho aleatório (obtido de uma distribuição normal).

Na avaliação dos indivíduos, o *fitness* é a exatidão de validação, sendo treinados, cada um, até atingir um certo limiar de melhoria. Só quando a melhoria for mínima, o indivíduo para de ser treinado, garantindo que é justamente avaliado e mantendo o treino eficiente, já que haverá um número limite de épocas.

É necessário colocar todos os indivíduos que participam da avaliação (o indivíduo a ser avaliado, o melhor individuo local e global), então os melhores indivíduos são redimensionados e só depois o individuo é avaliado.

No fim da procura, a melhor arquitetura é treinada por 100 e 250 épocas no FashionMNIST e CIFAR10, respetivamente.

Os resultados mostraram que as arquiteturas obtidas são similares a uma arquitetura VGG e que usam a estratégia de aumentar o número de *kernels* nas camadas inferiores e diminuir nas superiores. Observa-se que os resultados são iguais ou superiores ao estado da arte, minimizando o consumo de tempo.

Assim, o que permite otimizar e diminuir o tempo de procura, é a limitação do espaço de procura e o número baixo de épocas. O autor, após ser contactado, afirmou que utilizou apenas 1 GPU para cada evolução executada, tornando o artigo legível para comparações.

### A Memetic Algorithm for Evolving Deep Convolutional Neural Network in Image Classification

Neste artigo [41] utiliza-se um algoritmo memético, o que significa que a procura é totalmente local. Para gerar a população é definido um número aleatório de camadas convolucionais e de *pooling*, garantindo que o tamanho da população seja alcançado. Selecionam-se tantas camadas convolucionais quanto de *poolings*, colocando-se uma camada de *pooling* depois de cada convolução.

A representação de cada indivíduo é uma string binária, onde os hífens separam as conexões de cada nó. O número de números binários reflete o número de camadas do indivíduo. Os bits de um nó representam se o nó recebe *input* do nó anterior com esse índice (primeiro bit significa primeiro nó, segundo bit segundo nó e assim sucessivamente). Só os nós anteriores é que se podem conectar aos posteriores a eles.

A mutação e recombinação são ambas uniformes e a seleção é feita por torneio binário. No fim, das operações genéticas aplica-se um algoritmo de procura local, este vai gerar todas as permutações possíveis para um indivíduo e vai selecionar algumas delas. Essas permutações serão novos indivíduos que serão avaliados, se algum for melhor que o origina, então este é substituído.

O método de avaliação é o treinar cada arquitetura por poucas épocas, de forma a poupar tempo.

Os testes não foram feitos no dataset CIFAR10, mas sim no *Fashion*-MNIST e em outras variações mais complexas do MNIST (MBR, MBI, MRD, MRDBI).

Não houve comparações entre o estado da arte, optou-se apenas por comparar com o algoritmo EvoCNN [42].

A evolução termina se não houver melhorias nas últimas 8 gerações.

Utilizou-se um RTX 2080 Ti para executar o algoritmo evolucionário.

O tempo reduziu para menos de metade e a exatidão aumentou 0.33%. Considerando que no EvoCNN foram utilizadas 2 GTX 1080 Ti, a redução de tempo não foi impactada somente pela diferença de GPUs, mas sim pelo método de avaliação e pelo término precoce do algoritmo evolucionário face a estagnação dos indivíduos.

### Two-Level Genetic Algorithm for Evolving Convolutional Neural Networks for Pattern Recognition

Neste artigo [43] é utilizada uma estratégia de 2 fases, aplicadas em um algoritmo genético:

1. A população é treinada por poucas épocas de forma a reduzir tempo
2. A população é composta pelos melhores indivíduos da primeira fase e estes são treinados por mais épocas

Em ambas as fases há uma população e operadores genéticos.

A execução do algoritmo, na fase 1, é normal, utilizando seleção por torneio para criar a nova população e aplica os operadores de recombinação e mutação. No entanto, de $Q$ em $Q$ gerações é executada a fase 2, onde são selecionados os $N$ melhores indivíduos da primeira fase, tornando-se a população da segunda fase.

As modificações aplicadas nos indivíduos desta fase são refletidas diretamente nos mesmos indivíduos da fase 1, de forma providenciar *feedback* à primeira fase.

A evolução termina após um certo tempo ou número máximo de gerações.

A codificação é feita através de células, de tamanho variável, que contêm várias camadas (nós) e parâmetros tanto estruturais (número de blocos e número de camadas por célula) como de treino (*warmup* e *learning rate*). Cada camada também tem os seus parâmetros internos, como o número e tamanho dos *kernels*, tipo de junção quando há mais que um *input* etc.

As conexões de cada nó, com os *inputs,* é codificada de forma binária e só os nós anteriores ao nó atual podem ser seus *inputs*.

Em termos de operadores genéticos, utiliza-se a mutação aplicada aos nós, podendo adicionar ou remover um nó. Além da mutação aplicada aos parâmetros dos nós que altera todos os parâmetros aleatoriamente tendo em conta o conjunto de valores disponível. Já a recombinação pode ser aplicada a um nó aleatório entre 2 indivíduos,

pois mesmo que não sejam do mesmo tipo (convolução ou *pooling*) todos têm os mesmos parâmetros, mesmo que não se use alguns deles. Logo não há limitações na recombinação. No entanto, o número de *inputs* pode ser diferente tendo em conta a sua posição na célula, podendo apenas aplicar este operador em nós com posições iguais. Já quando se usa a recombinação de listas de nós, é necessário ter em atenção que um indivíduo pode conter mais camadas que o outro. Para não haver problemas, os nós em excesso têm uma probabilidade de 50% de serem adicionados à nova célula.

Em termos dos parâmetros estruturais e de treino também são recombinados, sendo escolhido um aleatoriamente em um dos indivíduos ou calculado um novo valor tendo em conta ambos os valores.

A função de avaliação utiliza o erro mínimo (sendo um problema de minimização), do *dataset* de validação, das últimas *épocas* e relaciona com o tempo de treino, prejudicando arquiteturas que demoram muito tempo a treinar. Haverá um *warmup* de *W épocas* com um *learning rate schedule* linear, permitindo aumentar o *learning rate* de 0 até ao máximo e fazer o caminho inverso. Após a execução do algoritmo evolucionário o melhor indivíduo é re-treinado durante 90 épocas (600 no *datatset* CIFAR10) e utilizando o *dataset* de treino e validação, sendo também aplicado *data augmentation*.

Este algoritmo mostrou os melhores resultados diante o estado da arte, selecionado pelos autores, tendo sempre uma exatidão superior a 90% em qualquer *dataset*. Teve uma diferença significativa no datatset MRDBI, comparativamente aos demais algoritmos.

Em termos de rapidez não foram apresentados muitos dados relativamente ao tempo e/ou *hardware* utilizado. No entanto, este algoritmo tem um espaço de procura vasto e implementa um mecanismo de treino rápido, garantindo a seleção dos melhores indivíduos, devido ao nível 2 de treino. Além de reduzir o tempo de treino, penalizando arquiteturas que consumam elevados tempos de treino.

### A transfer learning based evolutionary deep learning framework to evolve convolutional neural networks

Este algoritmo [44] faz uso da técnica de aprendizagem por transferência, o que significa que se vai obter um modelo válido para um dado *dataset* tendo este sido treinado em outros *datasets* de problemas similares.

Este subdivide-se em 2 estágios:

1. *Datasets* similares treinam um bloco de CNN, sendo este bloco modificado por um algoritmo evolucionário
2. Os blocos de CNN do estágio anterior são empilhados obtendo uma CNN mais complexa para ser utilizada no *dataset* que se deseje.

A estratégia evolucionária é o modelo *surrogate* (algoritmo mais simples, tipicamente de *Machine Learning*, de forma avaliar a qualidade de uma arquitetura sem a treinar) implementado em [9].

Os *datasets* utilizados foram o MNIST e *Fashion*-MNIST, sendo que o modelo final deve utilizar o *dataset* CIFAR10.

Os resultados mostraram que apenas 2 algoritmos do estado da arte, selecionado pelos autores, superaram este algoritmo, porém necessitaram de milhares de horas de execução e obtiveram modelos com um número elevado de parâmetros. Enquanto este algoritmo demorou apenas 40 horas e obteve um modelo com poucos parâmetros.

O que permite este algoritmo obter bons resultados é o facto de treinar, os diversos blocos da rede, em *datasets* mais simples, porém com o mesmo problema de classificação do *dataset* desejado. Desta forma, os pesos são otimizados para um mesmo problema mais rapidamente.

### Automated design of CNN architecture based on efficient evolutionary search

No [45] utilizou-se um algoritmo genético que utiliza blocos para codificar os indivíduos, sendo eles dos seguintes tipos:

- Convolucional: Tem 2 camadas convolucionais e uma *skip Connection*; Número de *kernels*: {64, 128, 256}; Tamanho do *kernel* e *stride* é de 3x3 e 1x1, respetivamente
- *Pooling*: Pode ser *maximum* ou *average pooling* com *kernel* e *stride* de 2x2 para ambas
- *Attention*: No fim de cada indivíduo há um módulo triplo de atenção.

O bloco de atenção permite capturar interações interdimensionais utilizando um conjunto de pesos.

Cada indivíduo é uma string de inteiros separados por hífens. Se o inteiro $\in$ {64,128,256} então tem-se uma camada convolucional, se for um valor real entre [0, 0.5) é uma *max pooling*, se for entre [0.5,1) é uma *average pooling*. Como pode-se constatar a codificação é feita a nível de camadas e não de blocos.

Na inicialização da população usa-se um número máximo e aleatório, de blocos, para cada indivíduo, sendo necessário que o número de camadas convolucionais esteja entre [5, 15] e de poolings entre [1, 5].

Os operadores genéticos são recombinação de 1 ponto e mutação de alteração nos parâmetros de uma camada aleatória ou de adição ou remoção de uma camada qualquer.

Para aliviar o custo computacional, foi implementada uma função de avaliação que não utiliza retro-propagação em grande parte da evolução, o que é denominado de modelo *surrogate*. Apenas as primeiras 3 gerações são avaliadas por treino normal e treino curto aplicado a cada indivíduo, fazendo uso da retro-propagação. As demais gerações utilizam os indivíduos dessas gerações passadas para treinar um algoritmo de floresta aleatória (algoritmo comum em *Machine Learning* que faz uso de árvores de decisão) que tem como atributo alvo as exatidões e como atributos as arquiteturas.

Desta forma é possível avaliar as arquiteturas posteriores comparando a sua topologia. Quanto mais idêntica for a uma topologia anterior, então mais provável obter uma exatidão idêntica.

Na última geração, as arquiteturas classificadas com a melhor exatidão são treinadas por mais tempo (350 épocas) utilizando, novamente, a retro-propagação.

Os resultados mostraram que o algoritmo se aproxima do estado da arte selecionado, seja de arquiteturas manuais como de algoritmos automáticos. Em termos de tempo, foi o mais rápido, exceto por um algoritmo, mas esse não consegue utilizar uma grande população nem procurar por um espaço tão vasto. No entanto, como este artigo baseia-se em blocos comuns, também não consegue encontrar novas arquiteturas que ultrapassem o estado da arte.

Verificou-se que o mecanismo de atenção tem um impacto significativo no aumento da exatidão. Também se verificou que o classificador de floresta aleatória faz classificações simples, restringindo a diversidade populacional, porém melhora significativamente o tempo de execução, como pode-se observar na comparação com o estado da arte apresentado no artigo.

### Surrogate-Assisted Particle Swarm Optimization for Evolving Variable-Length Transferable Blocks for Image Classification

Este algoritmo [9] utiliza um algoritmo PSO, por este ter um custo computacional reduzido. A composição dos indivíduos é de *Dense Blocks*, sendo esta uma topologia conhecida e com bons resultados na literatura de redes neuronais convolucionais. O objetivo é treinar blocos em vez de redes completas, já que tem um custo computacional inferior.

Os elementos do vetor/partícula são o *growth rate*, ou seja, o número de *kernels* de cada camada dentro do bloco. Caso uma camada tenha um certo valor especial, significa que ela não está disponível no bloco.

Para não estarem constantemente a treinar topologias iguais, é utilizado um modelo e datatset *surrogate*. O *dataset* para treinar o modelo *surrogate* é construído com base nos parâmetros do indivíduo, nos erros de treino e exatidões de validação. O modelo é uma SVM *(Support Vector Machine)* que é treinada muito mais rapidamente do que uma rede neuronal. Este modelo serve apenas para verificar se cada novo indivíduo é melhor que o melhor indivíduo global. Se sim, procede-se ao treino do mesmo, se não

o *fitness* dele será zero. Só é verificado um indivíduo se a média de classificações do modelo *surrogate* seja superior a um limiar, caso contrário ele necessita de mais treino e de mais dados para treinar, logo é necessário avaliar, tradicionalmente, mais indivíduos.

O *dataset surrogate* é um subconjunto do conjunto de dados de treino (reduzido por um parâmetro, *data reduction*) que posteriormente, é aplicado um fator de *downsampling* para reduzir o tamanho das imagens e assim otimizar o treino. Este *dataset* é utilizado para treinar os blocos e avaliá-los.

Os treinos são feitos até um número baixo de épocas, sendo esse número o suficiente para estimar o quão boa é uma arquitetura.

No fim, o bloco obtido é concatenado algumas vezes e treinado o modelo final.

Aqui a ideia de aprendizagem por transferência também está presente neste artigo, sendo o objetivo destes blocos serem transferíveis para outros datasets similares. Por exemplo, treinar no CIFAR10 e utilizar os blocos treinados no CIFAR100.

Os resultados demonstram que o algoritmo obteve bons resultados comparativamente ao estado da arte, principalmente em termos de tempo de execução. Ao aproveitar-se o melhor bloco obtido no CIFAR10, verificou-se o melhor resultado no CIFAR100, provando a assim a sua transferibilidade e obtendo o melhor resultado de entre os demais algoritmos.

A técnica de utilizar um modelo mais simples e estimar o quão melhor ele é comparativamente ao melhor modelo global, é uma técnica efetiva para evitar o treino excessivo de indivíduos, assim como reduzir o número de épocas, utilizar um *dataset* mais pequeno com imagens mais pequenas, assim como a transferibilidade do bloco final para outros *datasets*, contribui para um algoritmo de baixo custo e flexível, podendo-se utilizar o mesmo resultado em vários cenários. Contudo, o algoritmo foi executado em múltiplas GPUs com processamento em paralelo, logo não é um algoritmo válido para a revisão. Porém, a abordagem é diferente, já que utilização de um modelo *surrogate* permite reduzir bastante o tempo de execução, então este artigo será mantido.

### EvoDCNN: An evolutionary deep convolutional neural network for image classification

O algoritmo genético [46] utiliza uma codificação de tamanho fixo de blocos, sendo eles:

- 1 a 5 Blocos convolucionais: São compostos por uma convolução seguida de *batch normalization* e uma *pooling*, havendo múltiplos parâmetros a serem otimizados como o número de *kernels*. Pode também conter um *dropout* (camada que permite apenas alguns *inputs*, deixando os restantes a zero).

- 1 Bloco de classificação: Pode ser uma *Global Pooling* (reduz o tamanho espacial da saída a 1) ou uma camada totalmente conectada (como as faladas no capítulo *de Deep Learning*). Este também contém parâmetros como o número de unidades e a possibilidade de *dropout*.
- 1 Bloco de treino: Contém os parâmetros do optimizador

Também é possível ter uma conexão residual entre o *input* e a última camada do bloco convolucional.

Para contornar a limitação de indivíduos de tamanho fixo, um valor binário é colocado no início, do bloco convolucional, de cada indivíduo, indicando se ele está ativo ou inativo.

Os operadores genéticos são constituídos por uma seleção por roleta, uma recombinação com 1 ponto e uma mutação que é aplicada aos dois novos indivíduos, sendo aplicadas 2 mutações a um e nenhuma a outro. As mutações são aplicadas, aleatoriamente, ao nível dos genes.

Apesar da população ser iniciada com 60 indivíduos, são só utilizados 30. Durante o treino é implementado um *early* stopping, para evitar continua caso não haja melhorias.

No fim, as 6 melhores arquiteturas são treinadas por 300 épocas.

Os resultados, comparado ao estado da arte selecionado pelos autores, este algoritmo foi o mais rápido, mantendo resultados inferiores, porém próximos dos melhores. Também é necessário considerar, que devido à granularidade dos indivíduos, obtém-se arquiteturas robustas, com elevado número de parâmetros, o que por conseguinte, impacta no tempo de execução.

O que permite melhorar o tempo de execução e diminuir o custo computacional é a utilização de blocos de camadas e da utilização do *early stopping*, o que evita treino desnecessário a indivíduos fracos. De uma forma mais abstrata, a seleção limitada de possíveis parâmetros para cada camada dos blocos, também ajuda a limitar o espaço de procura, porém considerando a vasta possibilidade de parâmetros aqui mostrada, esta existe esta vantagem.

### Particle Swarm Optimization for Efficiently Evolving Deep Convolutional Neural Networks Using an Autoencoder-based Encoding Strategy

Neste artigo [47] é pretendido implementar um PSO, onde são resolvidos alguns problemas comuns em utilizar este tipo de estratégia, como por exemplo, a questão do vetor (indivíduo) ter um tamanho fixo o que limita a construção de redes complexas.

Também pretende-se verificar o quão bem uma pequena amostra do *dataset* de treino, pode ser utilizada para inferir a performance de uma arquitetura, sendo que se constatou, anteriormente, que essa inferência pode causar imprecisão.

Para reduzir o custo computacional, utiliza-se um método de avaliação hierárquico que permite utilizar diferentes tamanhos do *dataset* de treino ao longo da evolução. Além de utilizar apenas um *Dense Block* que será posteriormente empilhado múltiplas vezes para construir a arquitetura final.

Como o PSO é um algoritmo aplicado num espaço contínuo e a representação do bloco é discreta, pois o *growth rate* (número de *kernels* por camada) assim como o número de camadas, são variáveis discretas. Foi, então, proposto um *autoenconder* (algoritmo que reduz o tamanho de um espaço) permitindo transformar um vetor de tamanho variável, representando um *Dense block*, para um vetor de tamanho fixo e com variáveis contínuas, desta forma, também se evita diferenças entre inteiros e entre camadas ativas e desativas. Há um tamanho máximo do vetor variável, sendo este definido com base nas condições de *hardware*, além de um tamanho mínimo para garantir que não há blocos demasiadamente simples.

Além do *encoder*, há também um *decoder* para descodificar o vetor contínuo. Este *autoenconder* faz uso de camadas totalmente conectadas e permite extrair informações do vetor discreto, traduzindo-o numa representação mais compacta e contínua, num novo espaço latente normalizado. Esta representação será utilizada pelo algoritmo PSO.

Antes da execução do algoritmo evolucionário, o *autoenconder* é treinado com pares de blocos aleatórios, de forma a minimizar o erro de decodificação e garantindo que há uma relação de similaridade de arquiteturas e escala, por isso os dados de treino são pares de blocos representados vectorialmente, com tamanhos distintos.

A função de custo para o treino do *autoencoder*, soma os resultados de várias outras funções de custo:

- Função de custo para similaridade de arquiteturas
- Função de custo para similaridade de escala
- Função de custo da reconstrução (uma para cada vetor do par)

Desta forma, blocos com arquiteturas e escalas similares devem-se manter com representações latentes similares.

A população é então inicializada com vetores discretos aleatórios que servem de treino para o *autoencoder* e, posteriormente, o espaço latente é utilizado como representação de cada indivíduo.

Para reduzir o custo computacional, no início da evolução, os indivíduos são treinados num pedaço do *dataset* de treino e os seus pesos são guardados, para cada indivíduo. No fim de cada treino, é obtida a exatidão de validação e utilizada como *fitness* do indivíduo. Após toda a população passar por esse treino, os 3 melhores indivíduos, tendo em conta a exatidão normal, são selecionados, e continuam o treino num *dataset* de treino maior. Cada indivíduo terá 2 exatidões: normal e a progressiva – sendo que a normal usa o treino inicial e a progressiva usa o treino com o *dataset*

maior, em que todos os indivíduos, exceto os 3 melhores, têm a exatidão progressiva igual a 0.

Em todas as avaliações, é utilizado um limiar para o número de épocas treinadas e para um mínimo ideal de erro, quando um for atingido, para-se a avaliação do indivíduo.

Este método de avaliação explora a poupança de recursos com treinos curtos no início, devido à variabilidade do espaço de procura inicial. O treino em *dataset* maior permite encontrar indivíduos que têm bons resultados num *dataset* maior, logo também o deverão ter em um mais pequeno.

O critério de paragem acontece quando a melhor partícula/indivíduo global não é atualizada em 5 gerações seguidas, obtendo-se assim a partícula com o melhor *Dense Block* encontrado.

Esse bloco será copiado para criar vários outros blocos iguais que serão empilhados com camadas de *pooling* entre eles. Para assim, reduzir em metade a dimensão espacial, havendo limite para o número deste tipo de camadas, já que a dimensão mínima do *feature map* é de 1x1. Para selecionar o número de blocos a serem empilhados utiliza-se um *grid search* (procura baseada num conjunto de possíveis valores).

A arquitetura encontrada para o *dataset* CIFAR10, foi transferida para o CIFAR100 e *ImageNet,* de modo a mostrar a transferibilidade. Assim o CIFAR10 mostrou o terceiro melhor resultado em termos de exatidão, considerando que as 2 metodologias que o ultrapassaram, demoraram mais tempo. No CIFAR100 obteve-se o melhor resultado comparativamente aos demais algoritmos, sendo o terceiro que demorou menos tempo. É necessário salientar que ao utilizou-se a arquitetura obtida para o CIFAR10 e apenas executou-se o *grid search* para escolher o número ideal de blocos para o CIFAR100, obtendo-se um resultado similar, mas em metade do tempo.

A transferibilidade também se mostrou vantajosa no *ImageNet*, mantendo resultados próximos do estado da arte, demorando significativamente menos tempo que os demais algoritmos.

Também se concluiu que o *autoencoder* encontra fortes correlações entre os vetores, podendo assim garantir a escala e similaridade de arquiteturas, mantendo a sua representação idêntica. Os resultados mostraram que utilizar um *dataset* mais pequeno para identificar os melhores candidatos e depois avaliá-los em um maior, para assim, realmente distinguir-se os melhores dos bons candidatos, permite resultados com um equilibro entre exatidão e tempo de execução.

Assim além de obter-se uma redução no custo computacional devido à partilha de pesos e à exploração de um novo método de avaliação que tira partido da utilização de *datasets* reduzidos e ao uso de um bloco específico, com resultados relevantes na literatura, para assim permitir uma exploração local e global mesmo de limitando o espaço de procura, obtém-se um modelo que permite resolver problemas mais complexos, mesmo sendo evoluído num *dataset* mais simples.

Apesar do artigo não especificar o número de GPUs, após contacto com o autor, o mesmo indicou que utilizou apenas 1 GPU, tornando este artigo válido para análise.

### Multi-Objective Evolutionary Search of Compact Convolutional Neural Networks with Training-Free Estimation

O artigo [10] propõe um algoritmo PSO, utilizando uma função multiobjectivo que pretende minimizar o custo de validação do modelo e garantir um número razoável de parâmetros, porém não demasiado baixo nem alto.

Os módulos convolucionais são do tipo *MobileNet* garantindo assim convoluções com menos parâmetros. Esse módulo também tem um módulo de atenção para os *kernels* dos *feature maps*.

Este algoritmo não treina os indivíduos, porque utiliza uma medida baseada num *neural tangent kernel* [48], permitindo avaliar a treinabilidade do modelo sem treiná-lo tradicionalmente via retro-propagação. Este obtém os valores próprios da matriz espectral do *kernel*, podendo-se fazer um rácio que será utilizado como valor de custo de treino, pois este valor está correlacionado com o custo de validação.

Utilizando a dominância de *Pareto* (mostra quais os agentes que têm valores próximos dos ideais para os objetivos), atualiza-se os melhores indivíduos (global e local) utilizando as seguintes funções:

- Minimização do custo de validação
- Contagem de parâmetros da arquitetura

Como menos parâmetros pode piorar a performance do modelo, optou-se por ter um parâmetro que define um limiar de custo de validação, removendo assim os indivíduos que tenham um valor superior. Caso ainda fiquem mais indivíduos daqueles que se pretendem, removem-se aqueles com menos parâmetros.

Os resultados, avaliando pelo grupo com mais parâmetros, mantiveram um número baixo de parâmetros e o tempo mais baixo comparativamente com os demais algoritmos. A taxa de erro foi próxima das mais baixas, em ambos os *datasets* testados.

Assim a utilização do *Neural Tangent Kernel* permitiu encontrar indivíduos com boa performance e baixo número de parâmetros. Sem este *kernel* não se obteve resultados tão promissores. Já a utilização do módulo *MobileNet* com o módulo de atenção, também apresentou melhores resultados do que apenas utilizando um simples bloco *MobileNet*. Após conversar com o autor, o mesmo afirmou que só utilizou uma GPU para os testes, sendo este artigo válido para análise.

### Designing Convolutional Neural Networks using Surrogate assisted Genetic Algorithm for Medical Image Classification

O algoritmo proposto em [49], é genético e utiliza células que podem ter diferentes operações, como convoluções (3x3 e 5x5) do tipo *separable* e *dilated*, além de blocos residuais invertidos (fazendo assim conexões residuais entre células) e camadas *pooling*.

Elas são codificadas num vetor de números reais, no intervalo [0,1], tendo cada operação um valor real atribuído. Para inicializar a população de uma forma bem distribuída ao longo do espaço de procura, utilizou-se a técnica *Latent Hypercube Sampling* (LHS). Já os operadores genéticos são:

- Seleção por torneio binária
- Recombinação com 1 e 2 pontos
- Mutação por troca aleatória (número real aleatório no intervalo real [0,1])

A função de avaliação utiliza o custo de validação como *fitness*, porém para minimizar o tempo e o custo computacional do algoritmo, apenas 20% dos indivíduos são avaliados treinando tradicionalmente com a retro-propagação, já os 80% são estimados através de um modelo *surrogate*. Este recebe os genótipos e os seus erros de validação e estima o erro de validação de outros genótipos. Ele é treinado com os dados obtidos pelos 20% dos indivíduos que são treinados e avaliados normalmente.

Comparando os resultados deste algoritmo, destaca-se que este foi o que demorou menos tempo e teve a arquitetura com menos parâmetros, mantendo resultados competitivos. Isto deve-se ao seu ao vasto espaço de procura, sem grandes limitações, exceto o número máximo de camadas por indivíduo e também ao *XGBoost*, o modelo *surrogate* escolhido como estimador, sendo um algoritmo parecido com a floresta aleatória e tende a obter melhores resultados.

### A Hybrid Search Method for Accelerating Convolutional Neural Architecture Search

Neste artigo [50] é apresentado um algoritmo genético que utiliza um modelo *surrogate* para minimizar o custo computacional da procura. O algoritmo divide-se em avaliação de baixa e alta-fidelidade, onde a baixa fidelidade utiliza o modelo *surrogate* e a de alta-fidelidade utiliza o treino tradicional das redes neuronais.

Inicialmente são amostradas um conjunto de arquiteturas, sendo elas totalmente treinadas e avaliadas. Posteriormente, o *surrogate* é treinado com a arquitetura codificada com o respetivo *fitness*, sendo este o atributo alvo.

Quando o *surrogate* estiver pronto, é executada uma exploração global, ou seja, são selecionadas algumas arquiteturas já amostradas anteriormente, são aplicados operadores genéticos nelas e é utilizado o *surrogate* para criar o seu *fitness*. As melhores arquiteturas são adicionadas a uma nova população juntamente com a antiga. No fim, apenas as N melhores arquiteturas são colocadas numa nova lista, repetindo-se o processo até alcançar o limite de gerações.

Esta nova lista será agora passada para a etapa de exploração local, onde a avaliação de alta-fidelidade ocorre, ou seja, as arquiteturas são totalmente treinadas e avaliadas. A melhor será tratada como pai e novas arquiteturas serão mutadas a partir dela (com adição ou remoção de camadas convolucionais), sendo estas avaliadas pelo *surrogate*. As melhores arquiteturas serão utilizadas como filhos para serem treinadas. O pai é substituído se um dos filhos for melhor que ele. Repete-se o processo até alcançar o limite de gerações.

A nova população criada é utilizada para otimizar o modelo *surrogate* e a melhor arquitetura é colocada na lista da exploração global para garantir que se gerem melhores arquiteturas.

Para comparar os resultados obtidos utilizou-se o *NAS-Bench-101* e *NAS-Bench-201*, onde constam arquiteturas e os seus resultados de diversos algoritmos de NAS (*Neural Architecture Search*).

Este algoritmo híbrido, mostrou o melhor resultado no CIFAR10 e resultados relativamente próximos nos restantes *datasets* comparativamente com os demais algoritmos, utilizando o menor tempo de procura.

Também foi mostrado que não ter exploração local impacta diretamente na exatidão das arquiteturas, sendo que o *surrogate* não tem novos dados para treinar. Não fazer re-treinos no modelo *surrogate* mostrou também piores resultados, seguido de não atualizar a população da exploração global após uma exploração local.

Assim conclui-se que a estratégia de utilizar um modelo *surrogate*, neste caso um MLP, de forma a evitar treinos das redes neuronais convolucionais, permite diminuir o tempo de execução, pois com apenas uma GPU, treinar 400 candidatos demoraria muito mais.

Tabela 2 - Dados de treino da rede neuronal (durante o algoritmo evolucionário)

| Artigo | Representação | Optimizador | Épocas | *Learning Rate* | Observações |
|---|---|---|---|---|---|
| [37] | Algoritmo Genético (Lista) | *Adam* | 4 | 0.001 | - |
| [38] | PSO | *SGD, Nesterov momentum* = 0.9, *L2 regularization* = 0.0001 | 1 | 0.1 | *Learning Rate Schedule Cosine Annealing,* *init* = 0.1 final = 0.0001 |
| [39] | PSO | *SGD, Nesterov momentum* = 0.9, *L2 regularization* = 0.0001 | 1 | 0.1 | *Learning Rate Schedule Cosine Annealing,* *init* = 0.1 |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | final = 0.0001<br>*Data Augmentation* [51] |
| [40] | PSO | - | *Fashion*-MNIST: 3<br>CIFAR10: 10 | - | - |
| [41] | Memético *(string binária)* | - | 1 ou mais | - | - |
| [43] | Algoritmo Genético (células) | *Adam* | Nível 1: 18<br>Nível 2: 54<br>Nível 1 CIFAR10: 36 Nível 2 CIFAR10: 108 | - | *Learning Schedule* Linear |
| [44] | PSO | - | - | - | - |
| [45] | Algoritmo Genético (*String*) | *SGD, momentum*=0.9 | 10 | 0.1 | - |
| [9] | PSO | *Adam* | 10 | - | - |
| [46] | Algoritmo Genético (blocos) | - | 20 | - | - |
| [47] | PSO | SGD | Básico: 60<br>Progressivo: 50 | - | - |
| [10] | PSO | - | - | - | - |
| [49] | Algoritmo Genético (listas) | - | - | - | - |
| [50] | Algoritmo Genético | - | - | - | - |

Tabela 3 - Dados do algoritmo evolucionário

| Artigo | Gerações | População |
|---|---|---|
| [37] | - | 2 |
| [38] | 100 | 50 |
| [39] | 100 | 50 |
| [40] | 20 | 20 |
| [41] | 50 | 50 |
| [43] | 20 | 20 |
| [44] | 50 | 30 |
| [45] | 10 | 40 |
| [9] | 50 | 30 |
| [46] | 12 | 30 |
| [47] | - | 30 |
| [10] | 40 | 40 |
| [49] | - | - |
| [50] | Global: 30<br>Local: 10 | Global: 30<br>Local: 5 |

Tabela 4 - Resultados e especificações do hardware utilizado

| Artigo | *Dataset* | GPU | Número Parâmetros | *GPU Days* | Exatidão |
|---|---|---|---|---|---|
| [37] | CIFAR10/100 | Nvidia K40 | - | 1.5 | CIFAR10: 85%<br>CIFAR100: 61% |
| [38] | CIFAR10/100 | Nvidia GTX 1080Ti | - | CIFAR10: 1.48<br>CIFAR100: 1.5 | CIFAR10: 95.22%<br>CIFAR100: 74.58% |
| [39] | CIFAR10 | Nvidia GTX 1080Ti | - | 1.5 | Local Best: 95.61%<br>Lookup Table*:* 95.73% |
| [40] | CIFAR10/*Fashion*-MNIST/MRDBI | - | CIFAR10: 0.7M<br>Fashion- | CIFAR10: 1.65 | CIFAR10: 93.72%<br>*Fashion*-MNIST: 95.07% |

| | | | | | |
|---|---|---|---|---|---|
| | | | MNIST: 0.53M | | MRDBI: 89.83% |
| [41] | *Fashion*-MNIST/MRDBI | Nvidia RTX 2080 Ti | Fashion-MNIST: 1.24M | *Fashion*-MNIST: 1.03 | *Fashion*-Mnist: 96.94% MRDBI: 84.08% |
| [43] | *Fashion*-MNIST/CIFAR10/MRDBI | - | - | MRDBI: 0.26 | MRDBI: 93.67% *Fashion*-MNIST: 95.44% CIFAR10: 96.05% |
| [44] | CIFAR10 | - | CIFAR10: 2.29M | CIFAR10: 1.7 | CIFAR10: 96.54% |
| [45] | CIFAR10/CIFAR100 | Várias RTX | CIFAR10: 2.8M CIFAR100: 3.4M | CIFAR10: 0.66 CIFAR100: 0.92 | CIFAR10: 95.78% CIFAR100: 77.02% |
| [9] | CIFAR10/CIFAR100 | Várias RTX 2070 | CIFAR10: 2.54M | CIFAR10: 3 | CIFAR10: 96.51% CIFAR100: 81.51% |
| [46] | CIFAR10 | Nvidia Tesla Volta V100 GPU | CIFAR10: 18.8M | CIFAR10: 7.8 | CIFAR10: 94.70% |
| [47] | CIFAR10/CIFAR100/*ImageNet* | Nvidia A6000 | CIFAR10: 3.6M CIFAR100: 5.42M ImageNet: 4.9M | CIFAR10: 3 CIFAR100: 4 ImageNet: 4 | CIFAR10: 97.49% CIFAR100: 83.83% ImageNet: 73.1% |
| [10] | CIFAR10/CIFAR100 | Nvidia A6000 | CIFAR10: 0.72M CIFAR100: 0.94M | CIFAR10: 0.2 CIFAR100: 0.2 | CIFAR10: 96.84% CIFAR100: 81.22% |
| [49] | CIFAR10 | Nvidia A100 | CIFAR10: 0.67M | CIFAR10: 0.8 | CIFAR10: 96.66% |
| [50] | CIFAR10/CIFAR100/*ImageNet* | Nvidia RTX 2080 Ti | - | 2.8 | CIFAR10: 94.07% CIFAR100: 72.55% *ImageNet:* 45.65% |

## 3.2. Técnicas de avaliação sem treino e *autoencoders*

Os artigos selecionados focaram-se em estratégias que implicam diretamente o funcionamento do algoritmo evolucionário e/ou a avaliação dos indivíduos (treino). No entanto, durante uma fase inicial do projeto, antes de se executar a procura por artigos, pensou-se em uma abordagem mais orientada aos dados. Isto deve-se ao facto

de que em áreas como ciência de dados, antes de se selecionar um algoritmo para extrair informações, é necessário trabalhar e tratar os dados. Em *Deep Learning*, considerando a complexidade dos dados, não é algo comum tentar entender e tratar os dados, porém há uma abordagem utilizada que visa simplificar os dados.

A abordagem em questão são os *autoencoders*, já mencionados anteriormente. Estes permitem aprender a função que representa os dados (normalmente, utilizando uma rede neuronal) permitindo assim desconstruir os dados, para uma representação latente e reconstruí-los novamente. Uma das aplicabilidades deste tipo de modelo é a redução de dimensionalidade [52], apesar de manterem o tamanho original dos dados, os valores dos pixéis, no caso de os dados serem imagens, são simplificados de forma a apagar certas arestas e características irrelevantes na imagem.

Um tipo comum de rede neuronal são as *Deep Belief Networks* (DBNs), estas utilizam *Restricted Boltzmann Machines* (RBMs) [53] sendo estas um tipo de *Markov Random Fields* (MRF), como mencionado em [54], permitindo assim estimar valor de um nó tendo o anterior e garante-se que há independência entre nós de entrada e do espaço latente (propriedades das cadeias de Markov). É uma rede com variáveis binárias, podendo ser observada como um grafo não direcionado pesado (dígrafo pesado), onde os nós de um lado contêm as entradas (dados), não tendo conexões entre si, e estes estão totalmente ligados aos nós do outro lado do grafo, sendo estes os nós que estimarão e reconstruirão os dados. Como pode-se observar na Figura 12, $v$ são os nós de entrada (unidades visíveis), conectados por pesos, $W$, aos nós $h$ (unidades escondidas).

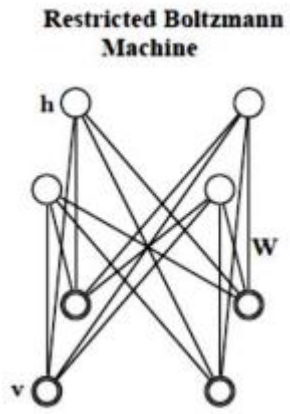


Figura 12 - Exemplo de uma RBM (adaptado de [54])

A RBM é modelada pela função de energia, sendo esta uma função de probabilidade conjunta:

$$P(v,h) = \frac{1}{Z} e^{-(-v^T W h - a^T v - b^T h)} \quad \textbf{26}$$

com $a$ e $b$ sendo resíduos (comumente denominados de *bias*) de $v$ e $h$, respetivamente. Já $Z$ é a função que calculará todas as probabilidades para todos os pares $\{v, h\}$.

Pretende-se assim, maximizar esta probabilidade, pois ela calcula a energia entre as entradas e o espaço latente, quanto maior a energia, mais as suas distribuições se aproximam, logo mais a rede estará a aprender sobre as entradas. Também é utilizado o método de *Constrastive Learning* que utiliza 2 fases: positiva e negativa – para calculando as probabilidades condicionais entre a distribuição das entradas e a latente e vice-versa, além de calcular a diferença entre as distribuições de probabilidade. Isto, porque calcular o gradiente de distribuições conjuntas é intragável. Logo, o treino é rápido, exatamente por não utilizar otimização por gradiente.

Uma RBM, após treinar terá, em $h$, o espaço latente com os dados simplificados (redução de dimensionalidade [55]). No entanto, uma RBM pode ser insuficiente e por isso criaram-se as *Deep Belief Networks*, que permitem conectar múltiplas RBMs, que funcionam como uma rede neuronal direta e totalmente conectada. Assim as DBNs podem ser classificadas como aprendizagem não supervisionada, ao contrário das redes neuronais convolucionais que implementam aprendizagem supervisionada, pois é lhes fornecido os resultados corretos de forma à rede aprender com os erros cometidos.

Considerando que os dados são imagens, simples multiplicações matriciais não são o indicado para extrair informações. Por isso, em [56], foi introduzida uma DBN convolucional (CDBN), onde fora criada uma *Convolutional* RBM (CRBM), utilizando convoluções em vez de multiplicações matriciais.

Além do artigo da construção de DBN convolucionais, encontrou-se outro artigo, onde é apresentada uma procura de arquiteturas de redes neuronais convolucionais, utilizando algoritmos evolucionários, porém sem treinar nenhum indivíduo. O artigo [11] explora as funções de ativação ReLU, de forma, a estimar se uma arquitetura de rede neuronal dará bons resultados ou não.

Ambos os artigos citados acima serão devidamente examinados, nos próximos subcapítulos.

### Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations

Como foi mencionado, RBM não é a melhor opção para aprender a distribuição de uma imagem, pois aplica transformações pixel a pixel. Em [56] introduz-se um novo tipo de RBM que utiliza convoluções, preservando assim os dados espaciais das imagens.

Enquanto, normalmente, as entradas são binárias, caso as imagens sejam a preto e branco, também é possível utilizar valores reais, no caso de as imagens serem coloridas. Para esse efeito, os dados das unidades são amostrados de uma distribuição normal.

As unidades visíveis, $v$, são a imagem, já as unidades escondidas, $h$, serão a distribuição estimada ou espaço latente. Contudo, para obter essas estimativas utilizar-

se-á convoluções e como fora constatado anteriormente, é útil ter múltiplos *kernels* para se extrair mais informação. Assim, as unidades escondidas terão profundidade, $K$, tendo-se $K$ $kernels$ aplicados a $v$.

Além da camada escondida, haverá uma camada de *pooling* para criar a invariância a pequenas translações e atuando como *downsampler* (diminuir o tamanho espacial), melhorando assim o *autoencoder*.

Considerando estas diferenças a função de energia será a seguinte:

$$E(v,h) = -\sum_{k=1}^{K} h^k \bullet (W^k * \mathrm{v}) - \sum_{k=1}^{K} b_k \sum_{i,j} h_{i,j}^k - a \sum_{i,j} v_{i,j} \qquad \textbf{27}$$

sendo • uma multiplicação elemento a elemento, seguida de somatório.

Para se gerar $h$ e $v$ alternadamente, utiliza-se a amostragem de *Gibbs*:

$$P\left(h_{i,j}^k = 1 \middle| v\right) = e^{-\left((W^k * v)_{ij} + b_k\right)} \cdot \sigma\left((W^k * v)_{i'j'} + b_k\right) \qquad \textbf{28}$$

$$P\left(v_{ij} = 1 \middle| h\right) = \sigma\left(\left(\sum_k W^k * h^k\right)_{ij} + a\right) \qquad \textbf{29}$$

sendo $\sigma$ a função de ativação sigmoide, $i'$ e $j'$ são as coordenadas para cada bloco utilizado na camada de *pooling*, posteriormente.

A camada de *pooling* é, normalmente, utilizada em redes neuronais diretas, sem conexões para trás. Neste artigo, como a rede não é direcionada, permitindo conexões em ambas as direções, esta camada foi adaptada para funcionar com este tipo de rede. Assim, a camada de *pooling* utiliza um fator de redução, $C$, permitindo diminuir o tamanho espacial da imagem. Utiliza-se a seguinte fórmula, para calcular o tamanho do *kernel* da camada de *pooling*:

$$N_p = \frac{N_h}{C} \qquad \textbf{30}$$

com $N_h$ sendo o tamanho do *kernel* da camada $h$. Para amostrar os valores para a camada de *pooling,* utiliza-se:

$$P(p^k = 0|v) = \sigma\left((W^k * v)_{i'j'} + b_k\right) \qquad \textbf{31}$$

A DBN será um conjunto destas CRBM, em que cada uma será treinada individual e sequencialmente, de baixo para cima, sendo o *output* de uma o *input* da seguinte CRBM. Utilizar a camada de *pooling*, mostrou uma performance dez vezes mais rápida.

Os resultados mostraram que foi necessário utilizar um treino esparso [57], para a rede detetar os contornos nas imagens. No entanto, a CDBN (*Convolutional* DBN) com 2 camadas escondidas, treinada num dado *dataset*, foi utilizada num outro *dataset* diferente e mostrou resultados razoáveis, demonstrando que esta aprende representações gerais e de alto nível.

Também se constatou que camadas elevadas, na rede, aprendem atributos que distinguem as imagens considerando a sua classe. Já as primeiras camadas focam-se nos contornos. Observa-se assim, um comportamento similar a uma rede neuronal convolucional.

Considerando os resultados apresentados, constata-se que mais testes devem ser executados em outros *datasets* com imagens RGB (*Red Green Blue*), para que se possa avaliar a diferença de tempo que uma rede neuronal convolucional levará a convergir utilizando o *dataset* original e o *dataset* pré-processado por uma CDBN.

### Neural Architecture Search without Training

A abordagem apresentada em [11], avalia a sobreposição de ativações das entradas em várias camadas numa rede neuronal. Essas ativações podem ser transformadas em códigos binários, pois a função de ativação é uma ReLU.

Assim, tendo uma rede neuronal, cada ativação aplicada a cada unidade, denomina-se de indicador binário, podendo ser zero quando o valor é negativo ou um quando o valor é positivo. Este indicador multiplica pelo valor, da unidade, obtendo assim a ativação da unidade.

Uma entrada, neste caso uma imagem, terá múltiplas camadas com múltiplas unidades. Pode ser tudo representado pelo código binário de cada indicador. Utilizando esse código, pode-se comparar duas imagens, medindo a distância de *Hamming* das suas ativações. Isto, porque quanto mais idênticas as ativações, menor a distância e pior será o resultado, já que a rede terá dificuldade em distinguir as duas imagens.

Baseando-se nesta premissa, pode-se criar uma matriz com as distâncias entre todas imagens de um lote de imagens. Redes com bons resultados, só têm alta similaridade na sua diagonal, pois a diagonal contém a distância de cada imagem para ela própria.

Para traduzir estes resultados para um número, sendo este posteriormente utilizado como medida de avaliação da rede, utiliza-se o determinante da matriz:

$$score = \log|K_H| \qquad \mathbf{32}$$

sendo $K_H$ a matriz com as distâncias.

Para testar esta metodologia, utilizaram-se os *datasets* para *benchmark* de NAS, como o *NAS-Bench-101, NAS-Bench-201*, *NATS-Bench SSS* e NDS. Estes contêm centenas

de milhares de redes neuronais treinadas com diferentes inicializações e em diferentes *datasets*, para que algoritmos de NAS possam obter exatidões e taxas de erro instantaneamente, sem a necessidade de treinar a rede. Também se podem utilizar para se obter arquiteturas que já tenham elevado desempenho, sendo esta ideia utilizada no artigo.

Obtiveram assim algumas arquiteturas sem qualquer treino, avaliando-as com a equação 32. Compararam, posteriormente, o *score* obtido com a exatidão de validação fornecida pelo *dataset* de *benchmark*. O resultado mostrou elevada correlação entre o *score* e a exatidão.

Outros testes foram executados e verificou-se que as imagens não influenciam os resultados, ou seja, a metodologia aqui aplicada captura informação sobre a arquitetura da rede e não sobre as imagens. As inicializações também se mostraram irrelevantes, assim como o tamanho dos lotes de imagens. Ao testar-se com redes treinadas por algumas épocas, concluiu-se que os *scores* são mais elevados, mas manteve-se a proporcionalidade, mostrando que treinar a rede não influencia o *score* calculado por esta metodologia.

Para mostrar na prática, implementaram um simples algoritmo para obter algumas arquiteturas, aleatoriamente, avaliá-las e escolher a melhor, repetindo o processo $N$ vezes. Os resultados das exatidões foram próximos dos algoritmos que utilizam treino, em todos os *datasets* (CIFAR10, CIFAR100 e *ImageNet-16-120*), porém apenas demorou 30 segundos.

O último teste, foi utilizar o algoritmo REA [58], porém ao selecionar os indivíduos iniciais, utilizou-se a metodologia da distância entre ativações para avaliar os indivíduos, mantendo só os $N$ melhores, executando o restante do algoritmo evolucionário. Ao limitar a execução por 12000 segundos, foi possível obter resultados idênticos ao algoritmo original.

Em suma, devem-se executar mais testes utilizando esta metodologia como método de avaliação, podendo assim utilizar algoritmos evolucionários com representações mais complexas, como árvores ou grafos, sem demorarem dias a executar e sem a necessidade de utilizar múltiplas GPUs.

## 3.3. Análise

Ao longo da análise individual de cada artigo, utilizaram-se múltiplos *datasets* para testar a performance de cada algoritmo proposto. Cada *dataset* será descrito nos próximos parágrafos de forma a explicitar a sua composição e complexidade.

O *dataset* CIFAR10 [59] é composto por 60 mil imagens RGB com tamanho 32x32. Tem 10 classes de objetos distintos com 6000 imagens para cada classe. Este é subdividido em 50 mil imagens para treino e 10 mil para teste.

O CIFAR100 [59] é igual ao CIFAR10, porém tem 100 classes, contendo 600 imagens de cada classe, tornando-o um desafio maior já que há pouca quantidade de imagens por classe.

O *Fashion*-MNIST [60] é um *dataset* que contém 60 mil imagens a preto e branco (grayscale) com tamanho 28x28. Contém 10 classes de objetos e também se subdivide em 50 mil imagens para treino e 10 mil para teste. Este pretende substituir o MNIST [61], mas ambos são utilizados para testar algoritmos de *Machine Learning*, não sendo tão comuns em *Deep Learning*.

O MRDBI [62] é uma variação do MNIST, onde os dígitos são rotacionados aleatoriamente e imagens de fundo são colocadas de forma a tornar o problema mais complexo.

O *ImageNet* [63] é um *dataset* que contém até 2 milhões de imagens com 1000 classes distintas, criado para a competição *Large Scale Visual Recognition Challenge* (LSVRC). As imagens neste *dataset* são mais próximas do mundo real, pois não têm tamanho ou qualidade garantida, ao contrário dos datasets anteriormente citados.

Os *datasets*, quando utilizados na prática, acabam por ser divididos em 3 subconjuntos:

- Treino: Estas são imagens utilizadas para treinar a rede neuronal
- Validação: Estas são as imagens utilizadas para testar as classificações da rede neuronal após um treino. Neste projeto, as exatidões de validação são vastamente utilizadas como valores de *fitness*
- Teste: Imagens utilizadas após o término de todas as repetições de treinos, para testar a exatidão e erro final das classificações da rede neuronal

Considerando agora os artigos analisados, verifica-se que a primeira relação que se pode observar- é o tempo de execução em função da exatidão, mas como nem todos os resultados apresentaram o tempo de execução, os datasets *Fashion*-MNIST e MRDBI não serão apresentados nesta relação.

A Figura 13, Figura 14 e Figura 15 mostram essa relação, onde cada ponto é um artigo rotulado com a sua referência. No CIFAR10 e CIFAR100 observa-se que o [37] apresenta resultados significativamente mais baixos, a nível de exatidão. A causa provável destes resultados é devido ao uso da estratégia (1+1), já que utilizar apenas 2 indivíduos limita o espaço de procura, mantendo uma procura local, pois só serão exploradas topologias próximas daquela que o primeiro pai tiver. Já [45], [10] e [49] são os algoritmos que têm tempo inferior a 1 dia e têm exatidões próximas de 96% no CIFAR10, apesar do [45] não poder ser justamente comparado, pois este utilizou múltiplas GPUs durante a sua execução. No entanto, os outros dois utilizaram apenas uma GPU, podendo-se observar que apesar de um ser um algoritmo genético e outro um PSO, ambos utilizam técnicas que evitam ou não treinam de todo a rede neuronal. Há 3 fatores que deixam o PSO há frente: ele ser um PSO (é eficiente na convergência),

não treinar os indivíduos e utilizar uma função multiobjectivo, o que pode retardar a procura, porém garante uma solução satisfatória, considerando as funções objetivo. EM [46] e o [50] também são utilizados algoritmos genéticos, tendo sido testados no CIFAR10, podendo-se observar que estão distantes tanto em tempo como em exatidão, comparativamente com os outros dois. No entanto, eles têm um espaço de procura mais vasto, fazendo com que seja necessário mais tempo para convergir, mesmo utilizando um modelo *surrogate*. O [50] tem testes nos 3 *datasets* devido à sua transferibilidade, porém a sua exatidão é mediana. No entanto, o [47] também foi testado nesses 3 *datasets*, demorando mais tempo, mas destacou-se sempre com a melhor exatidão. O tempo, possivelmente, é derivado do treino do *autoencoder* e das partículas. Porém o PSO com a possibilidade de representar partículas com tamanho variável, possibilitou que este obtivesse os melhores resultados.

Assim considerando os gráficos, o algoritmo com melhor equilíbrio é o [10]. O [47] mostrou resultados idênticos entre CIFAR100 e *ImageNet*, com mesmo tempo de execução, o que é impressionante, considerando a dificuldade deste último *dataset*. Logo a nível de robustez, ultrapassa o [10] em troca de mais algum tempo.

A GPU com mais FLOPS (*Floating-point Operations Per Second*) é a A6000, com 38 TFLOPS, podendo assim ajudar a obter melhores resultados, porém mesmo a Nvidia RTX 2080 Ti tem 13.35 TFLOPS, sendo improvável que essa diferença cause um impacto de 2.6 *GPU days*.

A média de *GPU days* e exatidão para os algoritmos genéticos é de 2.21 e 84.9%, respetivamentePara os algoritmos PSO tem-se 2.0 e 89.8%, respetivamente. Não é uma diferença significativa, já que não há muitos artigos, então não se pode afirmar que o PSO é mais rápido, mas a sua simplicidade ajuda a mantê-lo com resultados mais satisfatórios do que os algoritmos genéticos. Algoritmos genéticos tendem a ter mais dificuldade em fazer uma procura global, já que estes se baseiam em troca de genes ao longo de gerações, já o PSO permite uma procura mais global já que as partículas, mesmo dirigindo-se para um ponto em comum, cada uma tem a sua trajetória que é ajustada vagarosamente, o que possibilita encontrarem novas posições com melhores resultados, mudando assim a direção de todos. Isto permite que as demais partículas passem em outros pontos, enquanto mudam a direção. Este comportamento, garante uma melhor exploração do espaço de procura e por isso as arquiteturas encontradas são facilmente transferíveis para outros *datasets*, como em [9] e [47].

Assim, pode-se considerar que utilizar metodologias que evitem o treino das arquiteturas de redes neuronais, algoritmos PSO e representações que utilizam blocos conhecidos como o VGG, *Dense Blocks* ou *Mobile Net Blocks*, diminuem o tempo e garantem bons resultados. No entanto, ressalta-se que estimar exatidões com modelos *surrogate* ou outras metodologias livres de treino, pode causar imprecisões, assim como limitar o espaço de procura a certos hiperparâmetros de blocos conhecidos, fazendo com que não se encontrem novas arquiteturas ou até a melhor arquitetura para qualquer *dataset*. Contudo, também é possível observar que ao retirar esse

limitador, o tempo pode aumentar significativamente e os resultados podem não melhorar, como foi constatado em [46]. A utilização de mecanismos de atenção mostrou bons resultados, já que foram utilizados tanto em [45] como em [10],e mantendo bons resultados tanto no CIFAR10 como no CIFAR100. No entanto estes mecanismos tendem a ser dispendiosos a nível de recursos, logo é necessário utilizar técnicas que reduzam o tempo, como modelos *surrogate* ou estratégias *training free*.

Os resultados nos outros *datasets* (*Fashion*-MNIST e MRDBI) foram idênticos em todos os algoritmos, não contribuindo com informação útil.

Em suma, para criar um algoritmo evolucionário eficiente, de baixo custo, a nível computacional e em tempo de execução, mantendo exatidões e arquiteturas adaptáveis a múltiplos *datasets*, deve-se fazer uso de algoritmos PSO, utilizando técnicas para avaliar os indivíduos sem os treinar e utilizar blocos em vez de camadas separadas para a construção da arquitetura. Utilizar a partilha de pesos também ajuda em algoritmos que ainda necessitem de treinar alguns indivíduos.

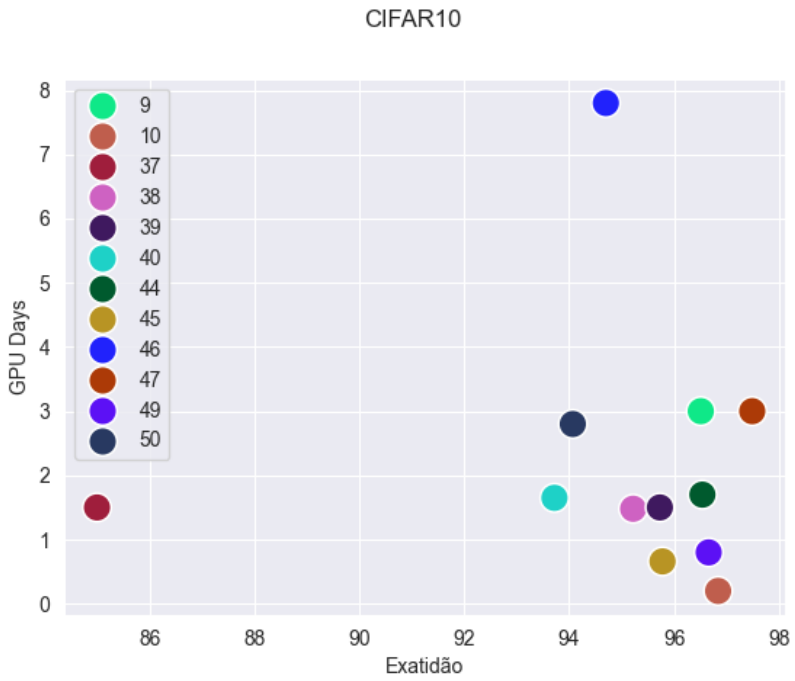


Figura 13 - Relação Exatidão-GPU Days no CIFAR10

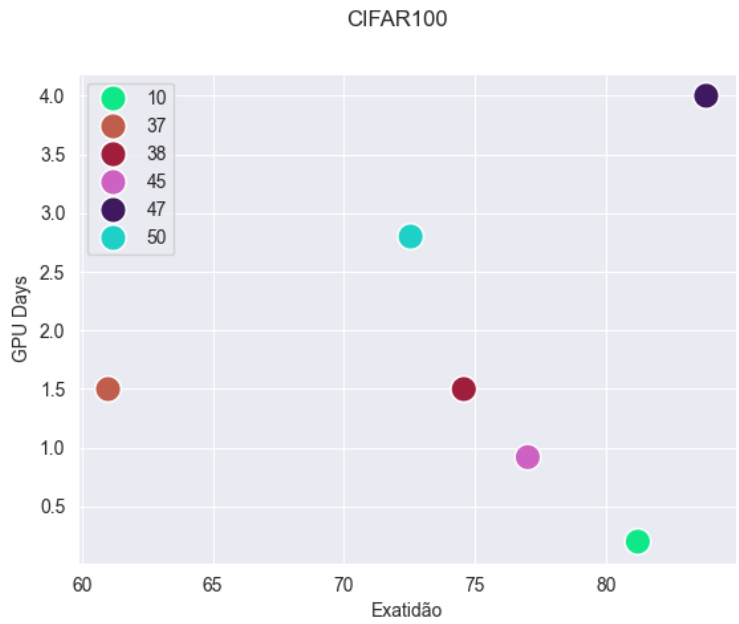


Figura 14 - Relação Exatidão-GPU Days no CIFAR100

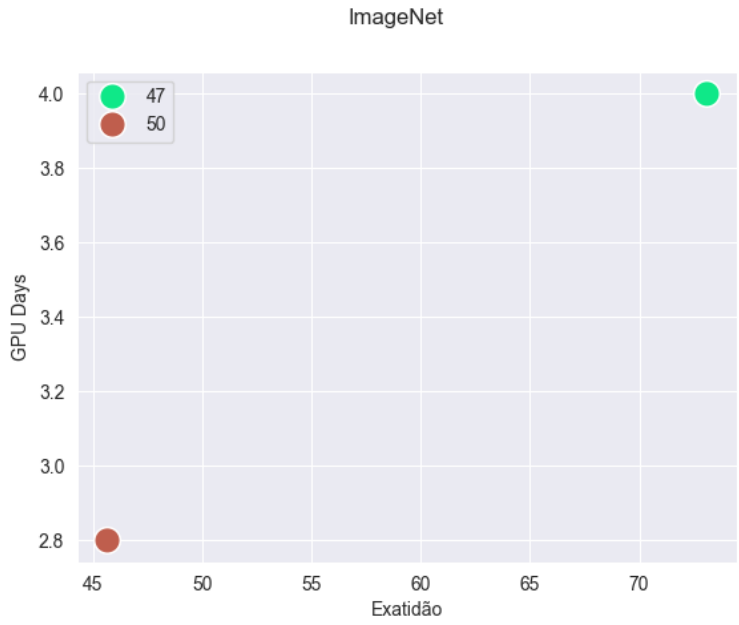


Figura 15 - Relação Exatidão-*GPU Days* no *ImageNet*

Após resumir e fazer a análise dos artigos, verificou-se que os melhores algoritmos utilizaram técnicas para minimizar ou remover por completo o treino das redes neuronais. Adicionalmente, um dos melhores algoritmos faz uso de um *autoencoder* para aumentar a capacidade representativa dos seus indivíduos.

Considerando estes aspetos, decidiu-se explorar utilização de um *autoencoder*, porém este será aplicado aos dadose não aos algoritmos evolucionários, permitindo assim que este se comporte como uma técnica de redução de dimensionalidade, podendo-se, posteriormente, otimizar a convergência das redes neuronais.

# 4. Casos de estudo

De forma a mostrar a performance do *autoencoder* retirado de [56] e verificar se o mesmo consegue aprender e reconstruir um dataset como o CIFAR10, implementou-se a CDBN. Ao implementar-se este cenário, concluir-se-á se se a abordagem deve ou não ser utilizada na próxima fase do projeto.

Para a implementação destas ideias foi necessário utilizar-se múltiplas ferramentas, como a linguagem de programação *Python* [64]. Esta é uma linguagem interpretada, orientada a objetos e tem uma implantação sólida no ramo da inteligência artificial, com acesso a múltiplas bibliotecas de alta performance desenvolvidas com linguagens como C++, de forma a serem eficientes, porém fáceis de utilizar devido à sintaxe simples do *Python*.

A biblioteca utilizada para a construção das redes neuronais e processamento dos datasets, foi o *Tensorflow* [65]. Inicialmente foi criado para ser uma biblioteca de cálculo numérico de elevada eficiência, porém tornou-se uma das mais populares bibliotecas de *Deep Learning*. Possui múltiplos módulos desde modelos pré-treinados em diversos tipos de *datasets* até módulos para processar imagens. E claro, oferece operações eficientes em *tensors* o que é uma mais-valia para redes neuronais. No entanto, o poder desta biblioteca está agregado ao poder de uma API (*Application Programming Interface*) de alto nível.

Essa API é o *Keras* [66], permitindo criar redes neuronais de forma simples, carregar e processar *datasets* de forma automática. O próprio *Tensorflow* depende muito desta biblioteca. Devido à sua simplicidade, é uma das ferramentas preferidas para se fazer a prototipagem de arquiteturas de redes neuronais.

Utilizar estas bibliotecas para treinar e inferir modelos de redes neuronais, é uma tarefa computacionalmente dispendiosa, logo um computador convencional não é o suficiente. Por isso, optou-se por utilizar uma plataforma na *cloud*, denominada de *Kaggle* [67]. Esta oferece competições na área de inteligência artificial e dispõe de *datasets* já carregados na plataforma para serem utilizados. No entanto, o benefício é a quota de horas semanais para utilização de GPU ou TPU (*Tensor Processing Unit*), sem qualquer custo. Para treinar rede neuronais profundas é necessário utilizar uma destas unidades de forma a minimizar o tempo de treino e inferência.

Para apresentar gráficos a utilização do *Matplotlib* [68] é inevitável, já que este *framework* permite facilmente construir gráficos utilizando as estruturas de dados do *Python* ou de outras bibliotecas como o *Numpy*.

O próximo subcapítulo visa mostrar a implementação do *autoencoder*, assim como os resultados obtidos.

## 4.1. Resultados da CDBN

A implementação da *Convolutional Deep Belief Network* foi baseada em [56] e [57], implementando-se assim a CDBN com os parâmetros de regularização aplicados às atualizações dos pesos.

Contudo, a CDBN foi desenvolvida como um classificador, ou seja, como uma alternativa a redes diretas e totalmente conectadas para classificação. Como se pretende utilizar estas redes para reconstruir a imagem, foram feitas algumas adaptações ao algoritmo.

Primeiramente, apesar de a rede aplicar o *Max Pooling*, a mesma utilizará uma camada convolucional transposta, denominada de *Deconvolution*, para desfazer a diminuição espacial feita pela camada de *pooling*. Assim mantém-se a invariância a pequenas translações, sem afetar o tamanho original da imagem.

A outra alteração é na saída da CDBN, onde, no artigo original, era retornado o *feature map* da última CRBM. Porém, para a reconstrução, é feita uma propagação para a frente, até à última CRBM. Após essa propagação, é executada outra, mas para trás, obtendo assim uma nova entrada reconstruída.

A Figura 16 apresenta o código da inicialização da CRBM, sendo instanciadas todas as variáveis necessárias, ou seja, *kernel* e os resíduos para a imagem e *kernel*. O *kernel* será amostrado de uma distribuição normal com média 0 e desvio-padrão 0.01, mas como é truncada, terá apenas valores positivos. A variável *sigma* conterá a variância, pois como os valores das imagens são do conjunto dos números reais, é necessário utilizar uma distribuição normal, sendo necessária uma variância. Também haverá variáveis para guardar valores anteriores e um *learning rate schedule*, sendo tudo utilizado na penalização dos pesos da rede. Todos os valores definidos, por omissão, são baseados em valores utilizados nos artigos e/ou valores que mostraram melhores resultados.

```python
class CRBM(tf.Module):
    def __init__(self,
                 visible_width,
                 visible_height,
                 visible_channels,
                 kernel_width,
                 kernel_height,
                 kernel_channels,
                 batch_size,
                 maxpooling = True,
                 sparsity_coef = 0.1,
                 sparsity_target = 0.1,
                 gauss_var = 0.2,
                 init_weight_std = 0.01,
                 learning_rate = 0.0001,
                 learning_rate_decay = 0.5,
                 momentum = 0.9,
                 decay_step = 50000,
                 weight_decay = 0.1
                ):

        self.visible_width = visible_width
        self.visible_height = visible_height
        self.visible_channels = visible_channels
        self.kernel_width = kernel_width
        self.kernel_height = kernel_height
        self.kernel_channels = kernel_channels
        self.hidden_height = visible_height
        self.hidden_width = visible_width
        self.batch_size = batch_size
        self.maxpooling = maxpooling
        self.sparsity_coef = sparsity_coef
        self.sparsity_target = sparsity_target
        self.gauss_var = gauss_var
        self.learning_rate = learning_rate
        self.learning_rate_decay = learning_rate_decay
        self.momentum = momentum
        self.decay_step = decay_step
        self.weight_decay = weight_decay

        # Criar variáveis
        self.kernels = tf.Variable(
            tf.random.truncated_normal(
                [kernel_height, kernel_width, visible_channels, kernel_channels],
                stddev=init_weight_std
            )
        )
        self.v_bias = tf.Variable(tf.constant(0.01, shape=[visible_channels,]), dtype=np.float32)
        self.h_bias = tf.Variable(tf.constant(-3.0, shape=[kernel_channels,]), dtype=np.float32)
        self.sigma = tf.Variable(tf.constant(gauss_var*gauss_var,
                                              shape=
[visible_height*visible_width*batch_size*visible_channels,]
                                             ),
                                 dtype=np.float32
                                )


        # Variáveis para guardar o estado anterior de outras variáveis
        self.kernels_prev = tf.Variable(
            tf.constant(0.0, shape=[kernel_height, kernel_width, visible_channels, kernel_channels]),
            dtype=np.float32
        )
        self.h_bias_prev = tf.Variable(tf.constant(0.0, shape=[kernel_channels,]), dtype=np.float32)
        self.v_bias_prev = tf.Variable(tf.constant(0.0, shape = [visible_channels,]),
dtype=np.float32)

        # Learning Rate Schedule exponencial para penalizar a aprendizagem dos pesos
        self.lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
            initial_learning_rate=self.learning_rate,
            decay_steps=self.decay_step,
            decay_rate=self.learning_rate_decay,
            staircase=True
        )
```

Figura 16 - Código CRBM

Na CRBM há um método que permite calcular os resultados da camada escondida, $h$, e da camada visível, $v$. A Figura 17 mostra esse método, o qual recebe a entrada, *layer*, e a direção para onde será feita a transformação.

Se for para a frente, *forward*, então a *layer* será a entrada que deve ser transformada para o espaço latente. Utilizando as probabilidades condicionais, obtém-se a camada no espaço latente, ficando armazenada na variável *bias*. No entanto, se o *Max Pooling* tiver ativo nesta CRBM, deve-se aplicá-lo, mas só se retorna o resultado se *result* não for *hidden*. Caso seja, então reverte-se o tamanho espacial, mantendo-se os resultados da *Max Pooling*. Se não houver *Max Pooling*, utiliza-se a probabilidade condicional comum das RBM, aplicando-se uma sigmoide.

Se for para trás, *backward*, inverte-se o *kernel* e aplica-se a probabilidade condicional. É importante ressalvar, novamente, que *gauss_var* é o desvio-padrão da distribuição de probabilidade dos valores da CRBM, sendo necessário dividir os resultados por ele quando a rede propaga o sinal para frente e multiplica-se quando é propagado para trás.

```
def infer_probability(self, layer, flow, result = "hidden"):
        if flow == "forward":
            # Transformar imagem na sua representação latente, h
            conv = tf.nn.conv2d(layer, self.kernels, [1, 1, 1, 1], padding='SAME')

            # gauss_var atua como desvio-padrão, fazendo parte do z-score aqui calculado
            conv = tf.divide(conv,self.gauss_var)

            # Aplicar resíduo
            bias = tf.nn.bias_add(conv, self.h_bias)

            if self.maxpooling:
                # Cria uma max pooling, com kernel e stride 2x2
                exp = tf.exp(bias)
                custom_kernel = tf.constant(1.0, shape=[2,2,self.kernel_channels,1])
                summ = tf.nn.depthwise_conv2d(exp, custom_kernel, [1, 2, 2, 1], padding='VALID')
                summ = tf.add(1.0,summ)

                # Define um kernel 2x2 para reverter para o tamanho espacial original
                ret_kernel = np.zeros((2,2,self.kernel_channels,self.kernel_channels))
                ret_kernel[:, :, range(self.kernel_channels), range(self.kernel_channels)] = 1

                custom_kernel_bis = tf.constant(ret_kernel,dtype = tf.float32)

                # Reverte o tamanho espacial com uma deconvolution
                sum_bis =  tf.nn.conv2d_transpose(
                    summ,
                    custom_kernel_bis,
                    (self.batch_size,self.hidden_height,self.hidden_width,self.kernel_channels),
                    strides= [1, 2, 2, 1],
                    padding='VALID'
                )

                # Retorna a imagem com o tamanho original
                if result == "hidden":
                    return tf.divide(exp, sum_bis)

                # Retorna a imagem com metade do seu tamanho
                return tf.subtract(1.0,tf.divide(1.0, summ))

            # Retorna imagem, sem max pooling aplicado
            return tf.sigmoid(bias)

        # Backwards, inverte-se operações e kernel para se obter a entrada reconstruída
        kernel_T = tf.transpose(tf.reverse(self.kernels,[True, False]),perm=[0, 1, 3, 2])
        conv = tf.nn.conv2d(layer, kernel_T, [1, 1, 1, 1], padding='SAME')
        conv = tf.multiply(conv,self.gauss_var)

        return tf.nn.bias_add(conv, self.v_bias)
```

Figura 17 - Calcular resultados da CRBM

Para a rede aprender é necessário que ela faça a amostragem de *Gibbs*, alternadamente, em ambas as direções. Normalmente, executar uma vez para cada direção é o suficiente. Na Figura 18 é apresentada essa amostragem, porém é necessário obter alguns valores aleatórios considerando tanto a distribuição empírica, $v$, como a distribuição a ser estimada, $h$. Como se está a utilizar uma distribuição normal, devido aos valores da imagem, o método *draw_samples* garante que os valores, após propagação para a frente, se mantêm positivos, sendo os negativos transformados em zero e os positivos em um. Tendo os valores importantes propagados, ao propagá-

los para trás, amostram-se novos valores dentro do intervalo da distribuição empírica, agora transformada pelo espaço latente.

```
def draw_samples(self, mean_activation, flow):

        if flow == "forward":
            return tf.where(

 tf.random.uniform([self.batch_size,self.hidden_height,self.hidden_width,self.kernel_channels]) -
mean_activation < 0,
                tf.ones([self.batch_size,self.hidden_height,self.hidden_width,
self.kernel_channels]),
                tf.zeros([self.batch_size,self.hidden_height,self.hidden_width,
self.kernel_channels])
            )

        mean = tf.reshape(mean_activation, [-1])
        dist = tfp.distributions.MultivariateNormalDiag(loc=mean, scale_diag=self.sigma)
        return tf.reshape(
            dist.sample(),
            [self.batch_size,self.visible_height,self.visible_width,self.visible_channels]
        )

def gibbs_sampling(self, data, num_gibbs = 1):

    v0 = data
    h0 = self.infer_probability(data,'forward')
    ret = self.draw_samples(h0, 'forward')

    for i in range(num_gibbs):
        vn = self.draw_samples(self.infer_probability(ret,'backward'), 'backward')
        hn = self.infer_probability(vn,'forward')
        ret = self.draw_samples(hn, 'forward')
    return v0,h0,vn,hn
```

Figura 18 - Amostragem de *Gibbs*

Para calcular o erro e atualizar os pesos utiliza-se o método de *Constrative Divergence*, implementado na Figura 19. Pode-se observar que antes de cada convolução, é sempre aplicado *padding* de zeros às entradas, mantendo assim o seu tamanho espacial após a operação. Este método de aprendizagem utiliza ambas as distribuições em dois estados distintos: inicial e após amostragem de *Gibbs*. Desta forma, permite verificar o quão elas se distanciam (sendo esse o erro calculado), atualizando os pesos para aproximá-las. Deve-se considerar que as atualizações dos pesos foram calculadas em lote, ou seja, consideraram-se múltiplas imagens durante todo o processo de aprendizagem. Logo, é necessário dividir as atualizações pelo número de imagens no lote, garantindo que a atualização é aplicada igualmente por cada imagem.

O método *apply_grad* aplica as atualizações, além de utilizar e atualizar os valores anteriores do *kernel* e resíduos. Já o *get_sparsity_penalty* aplica a regularização, como pode ser observado na Figura 20.

```
def do_contrastive_divergence(self, data, num_gibbs = 1, global_step = 0):
        v0,h0,vn,hn = self.gibbs_sampling(data, num_gibbs)

        # Padding de zeros para manter as dimensões após convolução
        v_pad = tf.pad(
            tf.transpose(v0,perm=[3,1,2,0]),
            [[0,0],
             [np.floor((self.kernel_height-1)/2).astype(int),np.ceil((self.kernel_height-
1)/2).astype(int)],
             [np.floor((self.kernel_width-1)/2).astype(int),np.ceil((self.kernel_width-
1)/2).astype(int)],
             [0,0]]
        )

        # Relacionar distribuição empírica e distribuição estimada inicialmente
        positive = tf.nn.conv2d(
            v_pad,
            tf.transpose(h0,perm=[1,2,0,3]), [1, 1, 1, 1], padding='VALID'
        )

        v_pad = tf.pad(
            tf.transpose(vn,perm=[3,1,2,0]),
            [[0,0],
             [np.floor((self.kernel_height-1)/2).astype(int),np.ceil((self.kernel_height-
1)/2).astype(int)],
             [np.floor((self.kernel_width-1)/2).astype(int),np.ceil((self.kernel_width-
1)/2).astype(int)],
             [0,0]]
        )

        # Relacionar distribuição empírica (já alterada) e distribuição estimada
        negative = tf.nn.conv2d(
            v_pad,
            tf.transpose(hn,perm=[1,2,0,3]), [1, 1, 1, 1], padding='VALID'
        )

        # Verificar o quanto se aproximaram uma da outra
        ret = tf.divide(positive - negative, self.gauss_var)

        # Aplicar atualização dos pesos e resíduos
        g_weight = tf.divide(tf.transpose(ret,perm=[1,2,0,3]),self.batch_size)
        g_weight_sparsity = self.get_sparsity_penalty('kernel', h0, v0)
        g_weight_l2 = tf.multiply(self.weight_decay,self.kernels)

        g_v_bias = tf.divide(tf.reduce_sum(tf.subtract(v0,vn), [0,1,2]),self.batch_size)
        g_v_bias = tf.divide(g_v_bias, self.gauss_var * self.gauss_var)

        g_h_bias = tf.divide(tf.reduce_sum(tf.subtract(h0,hn), [0,1,2]),self.batch_size)
        g_h_bias_sparsity = self.get_sparsity_penalty('hidden_bias', h0, v0)

        ret_w  = self.apply_grad(self.kernels, g_weight, self.kernels_prev, wd = True, wd_value =
g_weight_l2, sparsity = True, sparsity_value = g_weight_sparsity,  global_step = global_step)
        ret_bv = self.apply_grad(self.v_bias, g_v_bias, self.v_bias_prev, global_step = global_step)
        ret_bh = self.apply_grad(self.h_bias, g_h_bias, self.h_bias_prev, sparsity = True,
sparsity_value = g_h_bias_sparsity, global_step = global_step)
        cost = tf.reduce_sum(tf.square(tf.subtract(data,vn)))
        update = tf.reduce_sum(vn)

        return ret_w, ret_bv, ret_bh, cost, update
```

Figura 19 - *Constrative Divergence*

```
def apply_grad(self, parameter, grad_value, prev_value, wd = False, wd_value = None, sparsity = False, sparsity_value = None, global_step = 0):
        lr = self.lr_schedule(global_step)
        ret = tf.add(tf.multiply(self.momentum,prev_value),tf.multiply(lr,grad_value))
        ret = prev_value.assign(ret)

        if wd:
            ret = tf.subtract(ret,tf.multiply(lr,wd_value))

        if sparsity:
            ret = tf.subtract(ret,tf.multiply(lr,sparsity_value))

        return parameter.assign_add(ret)


def get_sparsity_penalty(self, name_type, h0, v0):
    ret = -2*self.sparsity_coef/self.batch_size
    ret = ret/self.gauss_var

    mean = tf.reduce_sum(h0, [0], keepdims = True)
    baseline = tf.multiply(
        tf.subtract(self.sparsity_target, mean),
        tf.multiply(tf.subtract(1.0,h0),h0)
    )

    if name_type == 'hidden_bias':
        return tf.multiply(ret,tf.reduce_sum(baseline, [0,1,2]))

    if name_type == 'kernel':
        v_pad = tf.pad(
        tf.transpose(v0,perm=[3,1,2,0]),
        [[0,0],
         [np.floor((self.kernel_height-1)/2).astype(int),np.ceil((self.kernel_height-1)/2).astype(int)],
          [np.floor((self.kernel_width-1)/2).astype(int),np.ceil((self.kernel_width-1)/2).astype(int)],
          [0,0]]
    )
        retBis = tf.nn.conv2d(
            v_pad,
            tf.transpose(baseline,perm=[1,2,0,3]), [1, 1, 1, 1], padding='VALID'
        )
        retBis = tf.transpose(retBis,perm=[1,2,0,3])

    return tf.multiply(ret,retBis)
```

Figura 20 - Gradiente e penalizações

Uma DBN utilizará múltiplas CRBM, além de poder repetir o processo de aprendizagem de cada CRBM. A Figura 21 mostra que o método *pretrain* implementa a rotina de treino de cada CRBM, garantindo que a CRBM seguinte recebe como entrada o resultado da CRBM anterior. Já os métodos, *encode* e *decode*, transformam e reconstroem, respetivamente, as imagens.

```python
import tensorflow as tf
from tqdm import tqdm

class CDBN(tf.Module):

    def __init__(self, batch_size, global_step = 1, **kwargs):
        super(CDBN, self).__init__(**kwargs)
        self.global_step = global_step
        self.batch_size = batch_size
        self.crbms = list()

    def pretrain(self, dataset):

        data = dataset

        for crbm in self.crbms:
            for i in tqdm(range(self.global_step)):
                for batch in data:
                    crbm.do_contrastive_divergence(batch, global_step=i)

            data = data.map(lambda x: crbm.infer_probability(x, "forward"))

    def encode(self, data):
        for crbm in self.crbms:
            h = crbm.infer_probability(data, "forward")
            data = h
        return h

    def decode(self, data):
        for crbm in self.crbms[::-1]:
            data = crbm.infer_probability(data, "backward")

        return data
```

Figura 21 - Código da DBN

Para testar-se a rede, utilizou-se o *dataset* CIFAR10, pois este é o mais simples dos três principais *datasets* utilizados neste projeto. A Figura 22 apresenta a imagem original e a sua reconstrução. É possível verificar que alguns pixéis se mantêm permitindo reconhecer a imagem, porém não é suficiente para afirmar que esta metodologia funciona.

Assim, criou-se uma rede neuronal convolucional simples, apresentada na Figura 23. Esta foi treinada e testada utilizando o *dataset* original e com o *dataset* transformado pelo DBN. O *dataset* de treino tem 50 mil imagens e o de teste 10 mil imagens e o treino foi executado por 10 épocas utilizando o optimizador *Adam*.

Com o *dataset* transformado os resultados forem idênticos aos do *dataset* original, porém com uma convergência mais rápida, ou seja, permitiram alcançar a mesma exatidão final em menos épocas. A utilização da DBN é assim uma boa técnica para minimizar o tempo de treino.

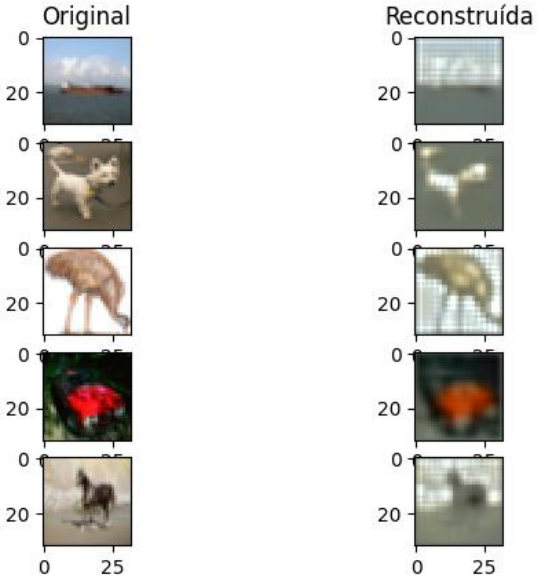


Figura 22 - Comparação do dataset original com o reconstruído

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 30, 30, 32) | 896 |
| max_pooling2d (MaxPooling2D) | (None, 15, 15, 32) | 0 |
| conv2d_1 (Conv2D) | (None, 13, 13, 64) | 18496 |
| max_pooling2d_1 (MaxPooling2D) | (None, 6, 6, 64) | 0 |
| conv2d_2 (Conv2D) | (None, 4, 4, 64) | 36928 |
| max_pooling2d_2 (MaxPooling2D) | (None, 2, 2, 64) | 0 |
| flatten (Flatten) | (None, 256) | 0 |
| dense (Dense) | (None, 64) | 16448 |
| dense_1 (Dense) | (None, 10) | 650 |

Total params: 73418 (286.79 KB)
Trainable params: 73418 (286.79 KB)
Non-trainable params: 0 (0.00 Byte)

Figura 23 - Rede neuronal convolucional simples

Uma DBN pode conter múltiplas CRBM e pode treiná-las por múltiplas épocas (denominadas no código por *global_step*). Constatou-se que utilizar muitas épocas prejudica a rede, por isso manteve-se apenas uma época, sendo esse o valor por omissão. Um número elevado de CRBM também prejudica a rede, logo decidiu-se utilizar apenas duas, como no artigo original.

No entanto, cada CRBM contém vários parâmetros, havendo assim diversas combinações possíveis. A Tabela 5 contém as possibilidades para cada CRBM, onde

cada parâmetro terá um total de: $2^3 = 8$ combinações possíveis. Como se vai utilizar duas CRBMS, então haverá $8^2 = 64$ combinações.

Tabela 5 - Combinações de parâmetros para a CRBM

| **Parâmetros** | **Valores** |
|---|---|
| Tamanho do *Kernel* | 3, 5 |
| Número de *Kernels* | 32, 64 |
| *Max Pooling* | *true*, *false* |

Considerando estas combinações, concebeu-se um DBN para cada e testou-se a performance da rede neuronal utilizando o *dataset* transformado pela DBN. Cada CRBM demorou cerca de 30 segundos a treinar, totalizando, 1 minuto de treino por DBN. A Figura 24 apresenta os resultados das três primeiras e últimas DBN, com base na exatidão de validação. As imagens a) e b) representam a exatidão e custo no *dataset* de treino, já as imagens c) e d) são relativas ao *dataset* de teste.

A melhor DBN está representada a tracejado, porém o resultado é idêntico às outras duas. Constata-se que a primeira CRBM deve conter 64 *kernel*s 3x3, aplicando a *Max Pooling*. Já a segunda CRBM deve conter *kernels* 5x5, mas tanto o número de *kernels,* como a utilização de *Max* Pooling, demonstraram ser irrelevante.

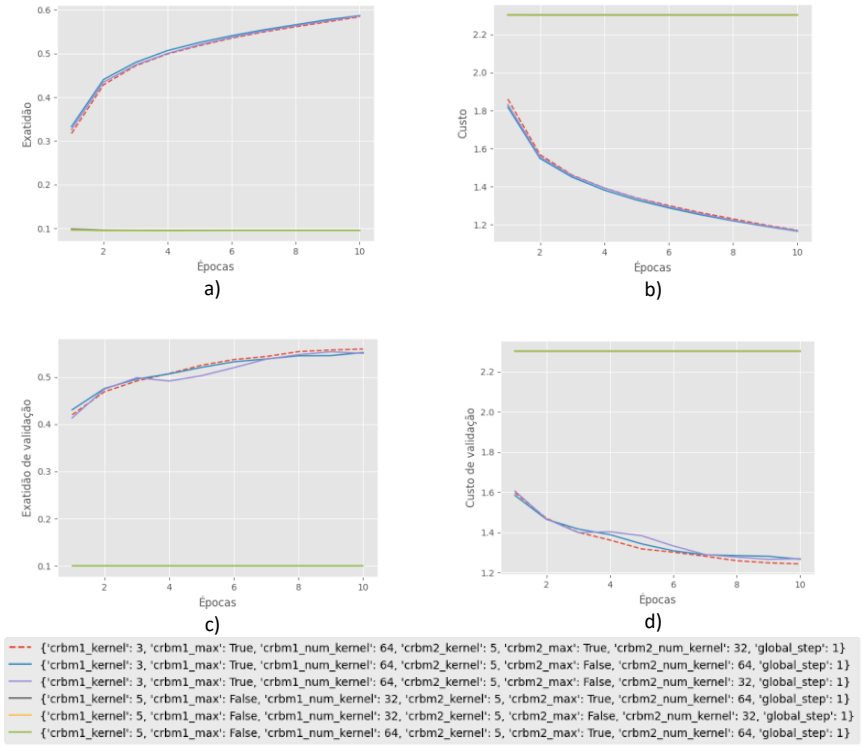


Figura 24 - Resultados do dataset transformado por diferentes DBN

Ao comparar-se os resultados do dataset transformado pela melhor DBN com os obtidos utilizando o dataset original, verifica-se, na Figura 25 que o dataset

transformado teve resultados significativamente piores. Já na Figura 26, observa-se que o tempo por época no dataset original é mais estável do que no dataset transformado, porém são tempos idênticos.

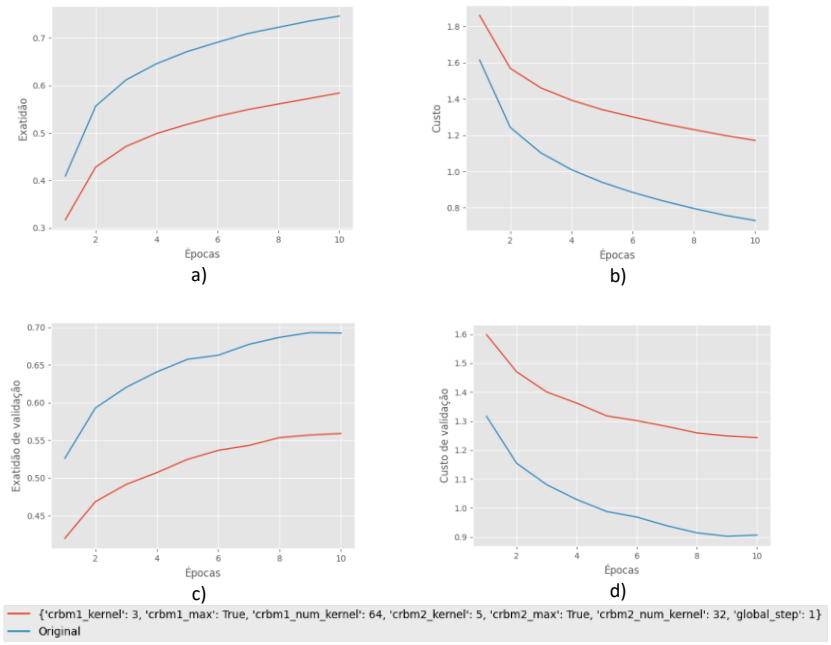


Figura 25 - Comparação entre melhor dataset transformado e o original

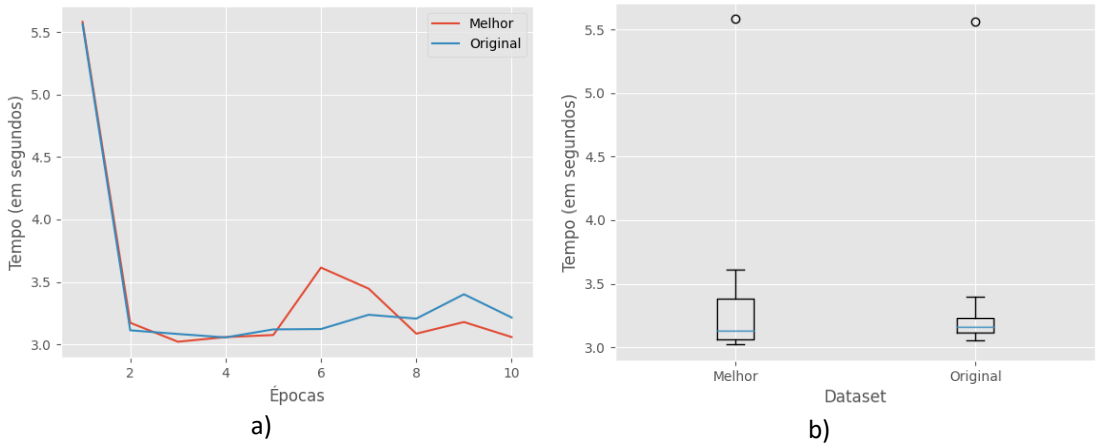


Figura 26 - Comparação de tempo de execução entre o dataset transformado e o original

Conclui-se que apesar desta rede ter o poder de reconstruir um *dataset*, mostra a sua ineficácia num simples *dataset* de 60 mil imagens RGB. Poder-se-ia melhorar os resultados, aplicando-se um treino baseado na descida do gradiente, porém isso impactaria significativamente no tempo de execução.

Desta forma, este algoritmo não será utilizado na segunda fase do projeto. Todo o código utilizado para construir, treinar e apresentar os gráficos, foi previamente disponibilizado *online* em [69].

# 5. Implementação

A principal questão que ficou por responder na sequência do trabalho realizado em projeto I foi: as métricas sem treino conseguem manter bons resultados quando utilizadas em conjunção com algoritmos evolutivos com representações mais complexas? Para responder a esta questão foram escolhidos dois algoritmos, ambos baseados em programação genética. A diferença entre programação e algoritmo genético é que, nos algoritmos genéticos, os indivíduos são representados por listas/*strings* de inteiros ou bits, enquanto a programação genética evolui programas de computador, representados por árvores, gramáticas ou DAG.

O *Fast-Denser++* [70] é um dos algoritmos de programação genética que foram selecionados. Este utiliza gramáticas para criar os seus indivíduos. A sua representação é denominada de DENSER (Deep Evolutionary Network Structured Representation), sendo esta uma lista de camadas que contém outras listas de hiperparâmetros.

O segundo algoritmo é o CGP-CNN (*Cartesian Genetic Programming-Convolutional Neural Network*) [71] que utiliza a representação CGP (*Cartesian Genetic Programming*) permitindo criar DAGs que representam a topologia de cada indivíduo. A seleção dos mesmos foi feita após procurar por algoritmos genéticos, tendo como objetivo obter algoritmos com representações complexas e distintas.

Estes algoritmos foram, posteriormente, modificados, de forma a utilizar as métricas sem treino, apresentadas anteriormente, sendo elas o NASWOT (*Neural Architecture Search Without Training*) e o NTK (*Neural Tangent Kernel*). Para clarificar o funcionamento de cada algoritmo, cada um deles será explicado de seguida.

## 5.1. Fast-Denser++

O *Fast-Denser++* [70] utiliza um método de *Neural Evolution* denominado de *DENSER* (*Deep Evolutionary Network Structured Representation*), baseando-se numa representação gramatical. A gramática livre de contexto, frequentemente representada no formato *Backus-Naur* [72], permite criar topologias de redes neuronais com um espaço de procura considerável, podendo ser alterada facilmente, pois tem um formato legível como pode ser observado na Figura 27.

A primeira regra, *<features>*, é a regra executada inicialmente, sendo que a mesma deve chamar outras regras, de forma recursiva, para assim criar uma topologia para uma rede neuronal convolucional. Além de regras, há terminais, sendo estes conjuntos de opções que terminarão a recursão durante a geração da topologia.

Este algoritmo também utiliza uma representação genotípica de dois níveis: externo e interno – externamente mostra o tipo da camada, ou seja, a regra escolhida na gramática, já na parte interna tem as opções escolhidas em todos os terminais referentes à regra selecionada. Estes níveis são representados por listas, sabendo-se o número exato de listas necessárias no nível interno, já que se sabe através da

gramática, a profundidade da recursão de cada regra. Quando é necessário escolher uma opção dentro de um conjunto, faz-se uma escolha aleatória. Assim, é possível descartar a operação modular executada para escolher opções nas regras e terminais, levando a topologias inacabadas.

De forma a facilitar o entendimento do genótipo, um exemplo do mesmo pode ser observado na Figura 28. O algoritmo também faz a procura não só pelo *feature extractor* (camadas de convolução e *poolings* que extraem informações das imagens), mas também encontra o número de camadas densas ótimo para encontrar relações nas informações do *feature extractor* e otimiza o próprio classificador, selecionando o optimizador e os seus parâmetros.

O operador genético de recombinação é de um ponto de corte e pode ser aplicado na parte externa e interna. Já o operador de mutação também pode ser aplicado internamente e externamente, removendo e adicionando camadas ou alterando os valores internos de uma camada.

É importante considerar que estes operadores são aplicados, independentemente, em todos os módulos (*feature extractor*, *camadas densas* e classificador), não sendo possível uma camada de um módulo passar para outro módulo. Este algoritmo foi originalmente testado pelos autores, no *dataset* CIFAR10, apresentando a exatidão de 89,44%, demorando 2,3 dias numa única GPU NVidia 1080 Ti.

```
<features> ::= <convolution> | <convolution>
               | <pooling> | <pooling>
               | <dropout> | <batch-norm>
<convolution> ::= layer:conv [num-filters,int,1,32,256] [filter-shape,int,1,2,5]
               [stride,int,1,1,3] <padding> <activation> <bias>
<batch-norm> ::=layer:batch-norm
<pooling> ::= <pool-type> [kernel-size,int,1,2,5]
               [stride,int,1,1,3] <padding>
<pool-type> ::= layer:pool-avg | layer:pool-max
<padding> ::= padding:same | padding:valid
<classification> ::= <fully-connected> | <dropout>
<fully-connected> ::= layer:fc <activation>
               [num-units,int,1,128,2048 <bias>
<dropout> ::=layer:dropput [rate,float,1,0,0.7]
<activation> ::= act:linear | act:relu | act:sigmoid
<bias> ::= bias:True | bias:False
<softmax> ::= layer:fc act:softmax num-units:10 bias:True
<learning> ::= <bp> <early-stop> [batch_size,int,1,50,500]
               | <rmsprop> <early-stop> [batch_size,int,1,50,500]
               | <adam> <early-stop> [batch_size,int,1,50,500]
<bp> ::= learning:gradient-descent [lr,float,1,0.0001,0.1]
               [momentum,float,1,0.68,0.99] [decay,float,1,0.000001,0.001]
               <nesterov>
<nesterov> ::= nesterov:True | nesterov:False
<adam> ::= learning:adam [lr,float,1,0.0001,0.1] [beta1,float,1,0.5,1]
               [beta2,float,1,0.5,1] [decay,float,1,0.000001,0.001]
<rmsprop> ::= learning:rmsprop [lr,float,1,0.0001,0.1]
               [rho,float,1,0.5,1] [decay,float,1,0.000001,0.001]
<early-stop> ::= [early_stop,int,1,5,20]
```

Figura 27 - Exemplo de gramática livre de contexto no algoritmo *Fast-Denser++*. Adaptado de [70]

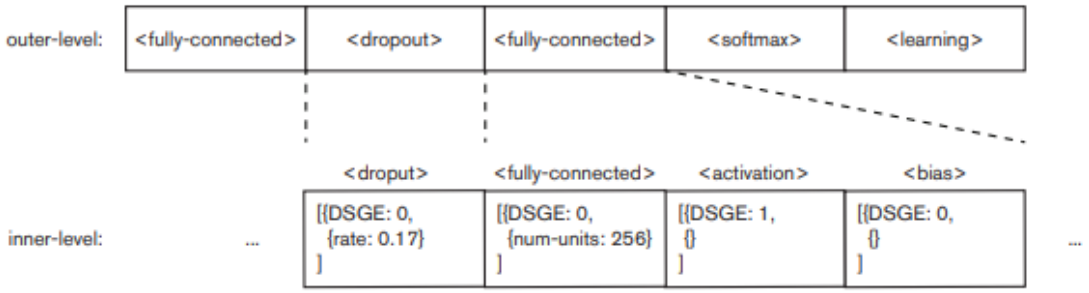


Figura 28 - Exemplo de genótipo no *Fast-Denser++*. Adaptado de [70]

## 5.2. CGP-CNN

O CGP-CNN [71] baseia-se numa representação de dígrafos acíclicos, DAG, sendo esta a melhor forma de representar uma rede neuronal que permite conexões residuais. No entanto, o grafo corresponde ao fenótipo, sendo o genótipo um conjunto aleatório de nós retirados de uma lista, sendo estes dispostos num plano cartesiano, como mostra a Figura 29.

Cada nó contém a sua informação a nível de aridade, ou seja, quantos *inputs* este pode receber. Por exemplo, um nó do tipo convolução recebe apenas uma entrada, enquanto um nó de soma ou concatenação necessita de duas entradas para poder executar a sua operação.

O CGP-CNN tem nós de convolução, *maximum* e *average poolings*, soma e concatenação, sendo que os nós de convolução têm variantes com 32, 64 e 128 número de *kernels* e *kernels* 1x1, 3x3, 5x5. Estes nós convolucionais são blocos compostos por Convolução, *Batch Normalization* e *ReLU*. Existe, porém, uma versão alternativa dos mesmos, com uma conexão residual entre o *input* e a saída do bloco.

Este algoritmo encontra apenas a topologia considerando o *feature extractor*, ou seja, o conjunto de camadas que extraem informações da imagem e propagam ao longo da rede. Não tem nem procura a parte de camadas densas que normalmente sucede o *feature extractor*, criando assim topologias com menos parâmetros e com menos tendência a sofrer *overffiting*.

As conexões são selecionadas aleatoriamente, porém têm em consideração o *levels-back*, sendo este um parâmetro que define quais nós podem servir de *input* para um nó mais à frente. Por exemplo, se o *levels-back* for igual a 1, então os nós da coluna 3 só podem ter *inputs* da coluna 2. Caso o *levels-back* fosse igual a 2, os *inputs* podiam ser da coluna 1 ou 2. Devido à aleatoriedade, pode haver nós inativos, ou seja, têm um *input*,

mas não são *input* de nenhum nó. Estes são removidos do fenótipo, mas mantêm-se no genótipo.

É também no genótipo que são aplicados os operadores genéticos, sendo que o CGP utiliza uma estratégia $1 + \mu$, onde um indivíduo é designado como pai inicialmente, e o mesmo é mutado $\mu$ vezes, criando $\mu$ filhos. Se algum deles for melhor que o pai, então este torna-se o novo pai. Pode-se constatar que não há necessidade de recombinação, pois a mutação já fará uma exploração local, além do próprio criador da representação CGP afirmar, em [73], que a recombinação não deve ser utilizada, porque não mostras resultados significativos.

Neste algoritmo utiliza-se uma mutação que altera, aleatoriamente, as conexões de um nó, mas há sempre probabilidade de modificar um nó inativo, pois o operador é aplicado no genótipo. Logo, criou-se uma mutação forçada que garante que pelo menos um nó ativo é modificado. Caso não se detete melhorias, o pai é mutado apenas nos nós inativos para aplicar *neutral drift* [74].

O número de filhos tende a ser pequeno, entre 2 e 10, logo produz uma população pequena, permitindo procuras mais rápidas sacrificando o tamanho do espaço de procura. O CGP-CNN foi originalmente testado nos *datasets* CIFAR10, utilizando blocos de convolução normais e residuais, obtendo exatidões de 93,25% e 94,02% respetivamente. Demorou 15,2 dias e 13,7 dias respetivamente, com 2 GPUs Nvidia Titan X.

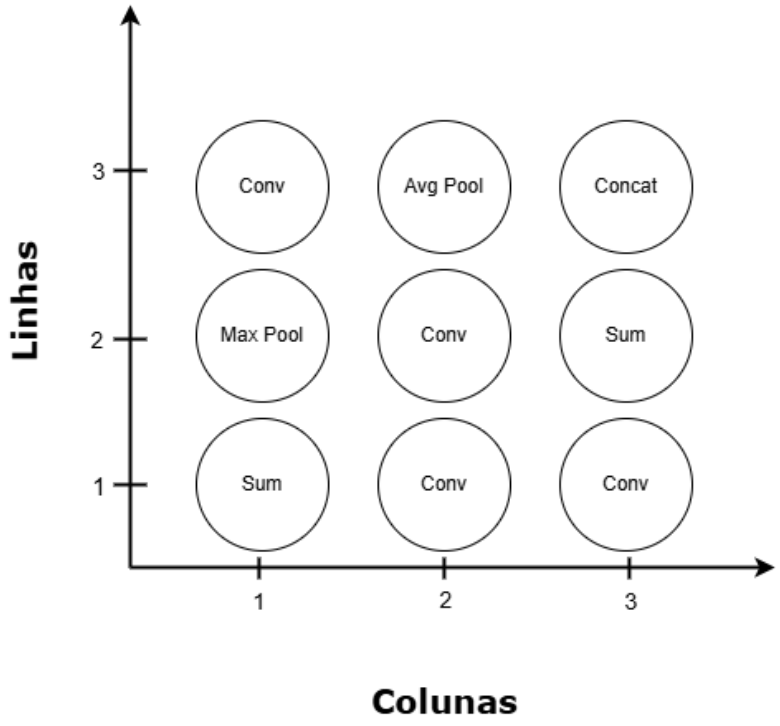


Figura 29 - Exemplo de genótipo no CGP-CNN

## 5.3. Integração das métricas de avaliação

Tendo sido introduzidos estes novos algoritmos e os seus resultados originais, prosseguiu-se com a adaptação das implementações dos mesmos para utilizarem as técnicas NASWOT e NTK já introduzidas anteriormente no capítulo Técnicas de avaliação sem treino e *autoencoders*.

### 5.3.1. Adaptação do algoritmo *Fast-Denser++*

O *Fast-Denser++* foi o primeiro algoritmo a ser modificado, utilizando a implementação do autor [75]. Começou-se por implementar as duas métricas de avaliação dentro do ficheiro *fitness_metrics.py*. Para o NASWOT foi implementada a função *relu_determinant*, (Figura 30 a), sendo que esta obtém as saídas de cada camada do modelo e calcula as ativações apenas para aquelas que têm função de ativação *ReLU*.

Após isso, é chamada uma função *calc_K*, que calculará o determinante da matriz das distâncias de *Hamming* entre cada imagem, utilizando o lote de imagens, *data*. Na Figura 30 b), é apresentada a função *calc_K,* a qual utiliza a multiplicação matricial entre a matriz *data* e a sua matriz transposta, somando com o complemento dessa multiplicação, obtendo assim a distância de forma eficiente.

```python
def relu_determinant(model, data):

    if isinstance(data, tuple):
        data = data[0]

    K_mat = tf.zeros((data.shape[0], data.shape[0]))

    relu_layers, layer_outputs = list(), list()

    for i, layer in enumerate(model.layers):
        layer_outputs.append(layer.output)

        if hasattr(layer, "activation") and layer.activation.__name__ == "relu":
            relu_layers.append(i)

    activation_model = tf.keras.models.Model(inputs=model.input, outputs=layer_outputs)
    outputs = activation_model(data, training=True)

    # Select only ReLU layers
    for i, layer in enumerate(outputs):
        if i in relu_layers:
            K_mat = calc_K(K_mat, layer)

    return K_mat
```

**a)**

```python
def calc_K(K_orig, data):
    data = tf.reshape(data, [data.shape[0], -1])
    x = tf.cast(tf.math.greater(data, 0), float)
    K_mat = tf.matmul(x, tf.transpose(x))
    K2_mat = tf.matmul(1.-x, 1.-tf.transpose(x))
    return K_orig + K_mat + K2_mat
```

**b)**

Figura 30 - Implementação NASWOT

Já para a implementação da métrica NTK, criou-se a função *ntk*, apresentada na Figura 31 a), onde é calculada a matriz contraída das matrizes jacobianas entre um lote de imagens, *x1*, e outro lote, *x2*, tendo em conta o modelo, *model*. A partir dessa matriz são obtidos os auto valores, obtendo o auto valor máximo e mínimo e, como resultado, o rácio entre eles.

A Figura 31 b) mostra a função *empirical_ntk_jacobian_contraction*, onde é calculada a contração das jacobianas (matrizes de gradientes, com base nos pesos) entre duas entradas distintas, camada por camada.

O método *tf.einsum* permite fazer multiplicações matriciais, explicitando as dimensões da matriz resultante e das matrizes de entrada. Fez-se uma soma, pela primeira dimensão, pois é nela que estão todas as contrações das jacobianas por camada. Já na Figura 31 c), a função *calc_eigen* calcula e encontra os máximo e mínimo auto valor da matriz anteriormente calculada. Estas duas funções estão anotadas com *@tf.function*, de maneira que o *Tensorflow* pode criar um grafo estático das operações, podendo assim otimizá-las.

```python
def ntk(model, x1, x2):
    result = empirical_ntk_jacobian_contraction(
        model,
        x1,
        x2
    )

    max_eig, min_eig = calc_eigen(result)

    return max_eig/min_eig
```

a)

```python
@tf.function
def empirical_ntk_jacobian_contraction(model, x1, x2):

    def compute_jacobian(model, x):
        # Jacobians should be done considering each layer (model.trainable_variables)
        with tf.GradientTape(persistent=True) as tape:
            tape.watch(model.trainable_variables)
            outputs = model(x)
        jac = tape.jacobian(outputs, model.trainable_variables)

        # Keep batch and the number of outputs (classes in the case of the CNN)
        return [tf.reshape(j, [j.shape[0], j.shape[1], -1]) for j in jac]

    jac1 = compute_jacobian(model, x1)
    jac2 = compute_jacobian(model, x2)

    # # Compute J(x1) @ J(x2).T
    result = tf.stack([tf.einsum('Naf,Mbf->NMab', j1, j2) for j1, j2 in zip(jac1, jac2)])
    return tf.reduce_sum(result, axis=0)
```

b)

```python
@tf.function
def calc_eigen(result):
    eigenvalues = tf.linalg.eigvalsh(result)
    return tf.reduce_max(eigenvalues), tf.reduce_min(eigenvalues)
```

c)

Figura 31 - Implementação do NTK

A métrica utilizada durante uma procura é explicitada no documento *config.json*, onde se deve colocar o nome da função a utilizar. Como estas métricas não utilizam treino e recebem argumentos como o objeto do modelo da rede e lotes de imagens, foi necessário alterar o método *evaluate* da classe *Evaluator*, onde é executado o treino de

cada candidato. Assim, na Figura 32 a), há uma condição que garante que se a métrica utilizada for NASWOT, então o treino consiste apenas em executar a função *self.fitness_metric* (armazenada num atributo do objeto).

Pode-se, também, observar que o lote de dados é extraído com base no tamanho do *batch_size* fornecido. No entanto, na Figura 32 b), quando a métrica é NTK, constata-se que é necessário utilizar um lote de apenas 32 imagens para garantir que o algoritmo não exceda a memória da GPU. Este valor foi obtido após alguns testes com vários tamanhos de *batch_size*.

Como o NTK é positivamente correlacionado com a taxa de erro de validação do modelo, então esta métrica deve ser utilizada para um problema de minimização. No entanto, o algoritmo de procura, *Fast-Denser++*, baseia-se num problema de maximização. Para tornar o resultado do NTK, ou seja, o valor de *fitness*, num problema de maximização, basta multiplicar o resultado por -1, de maneira que, quanto maior o valor negativo, maior e melhor o *fitness* do indivíduo.

```
517         if self.fitness_metric.__name__ == "relu_determinant":
518             if datagen is None:
519                 data = self.dataset['evo_x_test']
520                 data = data[:batch_size, :, :, :]
521             else:
522                 data = datagen_test.flow(self.dataset['evo_x_test'], batch_size=batch_size)
523                 data = next(iter(data))
524 
525             # Passing only a batch of data to evaluate
526             K_mat = self.fitness_metric(model, data)
527             _, det = tf.linalg.slogdet(K_mat)
528             accuracy_test = float(tf.get_static_value(det))
529 
```

a)

```
530         elif self.fitness_metric.__name__ == "ntk":
531             if datagen is None:
532                 data = self.dataset['evo_x_test']
533 
534                 # Not using batch-size for performance issues
535                 x1, x2 = data[:32, :, :, :], data[32:32, :, :, :]
536             else:
537                 data = datagen_test.flow(self.dataset['evo_x_test'], batch_size=32)
538                 x1, x2 = next(iter(data)), next(iter(data))
539 
540             # eig_val_ratio is positive correlated with error rate. Because we're maximizing fitness
541             # and we want to minimize eig_val_ratio , so we need to flip the sign
542             eig_val_ratio = self.fitness_metric(model, x1, x2)
543             accuracy_test = -float(tf.get_static_value(eig_val_ratio))
```

b)

Figura 32 - Função de avaliação dos indivíduos utilizando as novas métricas

### 5.3.2. Adaptação do algoritmo CGP-CNN

O próximo algoritmo implementado foi o CGP-CNN, sendo a implementação baseada no código dos autores presente em [76]. No entanto, a implementação original utiliza uma biblioteca alternativa ao *Tensorflow*, denominada de *PyTorch* e, por esse motivo, foi necessário adaptar todo o código do algoritmo, além de incluir as funções previamente descritas das métricas NTK e NASWOT.

A primeira modificação foi na classe *CgpInfoConvSet* encarregue de definir o número de linhas e colunas (nós) que existirão no plano cartesiano, assim como o parâmetro *levels-back*, encarregue de definir a distância que um nó pode estar de outro nó para ser elegível a fornecer *input*. A modificação, como se pode observar na Figura 33, consistiu em receber um parâmetro *set_type*, onde se define se o conjunto de nós é do tipo *ConvSet* ou *ResSet*.

A necessidade de alteração deve-se à forma como os autores testaram e apresentaram os resultados do algoritmo, tendo utilizado o seguinte conjunto de camadas: *ConvBlock*, *Maximum* e *Average Pooling*, *Concat* e *Sum* – definindo assim o conjunto *ConvSet*. Já o *ResSet* é definido pelas camadas: *ResBlock*, *Maximum* e *Average Pooling*, *Concat* e *Sum*.

A camada *ConvBlock*, como o nome indica é um bloco de camadas, tendo assim uma camada convolucional seguida de *Batch Normalization* e por fim uma camada com a função de ativação ReLU. Já a camada *ResBlock* é idêntica à *ConvBlock*, mas adiciona o *input* à saída da função de ativação, criando assim uma conexão residual. Também se pode constatar na Figura 33 que há vários tipos de blocos, mudando o primeiro e segundo número no seu nome. O primeiro número refere-se ao número de *kernels* e o segundo ao tamanho do *kernel* na camada convolucional.

```
class CgpInfoConvSet(object):
    def __init__(self, rows=30, cols=40, level_back=40, min_active_num=8, max_active_num=50, set_type="convset"):
        self.input_num = 1
        # "S_" means that the layer has a convolution layer without downsampling.
        # "D_" means that the layer has a convolution layer with downsampling.
        # "Sum" means that the layer has a skip connection.
        if set_type == "convset":
            self.func_type = ['S_ConvBlock_32_1',    'S_ConvBlock_32_3',   'S_ConvBlock_32_5',
                              'S_ConvBlock_128_1',    'S_ConvBlock_128_3',   'S_ConvBlock_128_5',
                              'S_ConvBlock_64_1',     'S_ConvBlock_64_3',    'S_ConvBlock_64_5',
                              'Concat', 'Sum',
                              'Max_Pool', 'Avg_Pool']
        else:
            self.func_type = ['S_ResBlock_32_1',     'S_ResBlock_32_3',    'S_ResBlock_32_5',
                              'S_ResBlock_128_1',     'S_ResBlock_128_3',    'S_ResBlock_128_5',
                              'S_ResBlock_64_1',      'S_ResBlock_64_3',     'S_ResBlock_64_5',
                              'Concat', 'Sum',
                              'Max_Pool', 'Avg_Pool']

        self.func_in_num = [1, 1,
                            1, 1, 1,
                            1, 1, 1,
                            2, 2,
                            1, 1]

        self.out_num = 1
        self.out_type = ['full']
        self.out_in_num = [1]
```

Figura 33 - Classe *CgpInfoConvSet* modificada

A classe *CNNEvaluation* também foi modificada, pois na sua versão original suportava processamento *multi-GPU*, algo desnecessário para esta implementação, onde apenas uma GPU é utilizada. Assim, a tarefa desta classe será apenas receber os dados e uma lista de genótipos de cada indivíduo e fornecê-los ao objeto da classe de avaliação. Essa classe é a *CNN_train* - esta carrega o *dataset* quando é instanciado um objeto desta classe e executa a avaliação de cada indivíduo quando o objeto é chamado.

A Figura 34 a) mostra o código original, onde é utilizada a biblioteca *PyTorch* para transformar e carregar as imagens. Deve-se notar que caso *validation* seja verdadeiro, o *dataset* para treino terá *data augmentation*, porque os autores utilizam o treino das redes neuronais para avaliar os indivíduos, logo é importante utilizar esta técnica para obter um maior número de dados. Caso seja para testar apenas o indivíduo obtido, obtendo o seu *fitness* final, então não há necessidade de *data augmentation*.

A implementação adaptada é apresentada na Figura 34, onde já se leva em consideração os *datasets* CIFAR10 e CIFAR100, pois ambos serão testados neste projeto e apesar de se manter a lógica de utilizar ou não *data augmentation*, esta será irrelevante, pois não se utilizará treino das redes neuronais para neuronais sua avaliação, mas sim as métricas NTK e NASWOT, onde apenas um lote de imagens será utilizado. Assim, manteve-se a implementação fiel à dos autores. A classe também recebe um novo parâmetro denominado de *eval_fn*, contendo a referência para a função da métrica a ser utilizada para avaliar cada rede neuronal.

```python
class CNN_train():
    def __init__(self, dataset_name, validation=True, verbose=True, imgSize=32, batchsize=128):
        self.verbose = verbose
        self.imgSize = imgSize
        self.validation = validation
        self.batchsize = batchsize
        self.dataset_name = dataset_name

        # load dataset
        if dataset_name == 'cifar10' or dataset_name == 'mnist':
            if dataset_name == 'cifar10':
                self.n_class = 10
                self.channel = 3
                if self.validation:
                    self.dataloader, self.test_dataloader = get_train_valid_loader(data_dir='./', batch_size=self.batchsize,
                                                                                   augment=True, random_seed=2018, num_workers=1, pin_memory=True)
                    # self.dataloader, self.test_dataloader = loaders[0], loaders[1]
                else:
                    train_dataset = dset.CIFAR10(root='./', train=True, download=True,
                            transform=transforms.Compose([
                                transforms.RandomHorizontalFlip(),
                                transforms.Scale(self.imgSize),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
                            ]))
                    test_dataset = dset.CIFAR10(root='./', train=False, download=True,
                            transform=transforms.Compose([
                                transforms.Scale(self.imgSize),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)),
                            ]))
                    self.dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=self.batchsize, shuffle=True, num_workers=int(2))
                    self.test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=self.batchsize, shuffle=True, num_workers=int(2))
                print('train num    ', len(self.dataloader.dataset))
                # print('test num     ', len(self.test_dataloader.dataset))
            else:
                print('\tInvalid input dataset name at CNN_train()')
                exit(1)
```

**a)**

```python
class CNN_train():
    def __init__(self, dataset_name, validation=True, verbose=True,
                 imgSize=32, batchsize=128, eval_fn=relu_determinant):
        self.verbose = verbose
        self.imgSize = imgSize
        self.validation = validation
        self.batchsize = batchsize
        self.dataset_name = dataset_name
        self.eval_fn = eval_fn

        # load dataset
        self.channel = 3
        if dataset_name == 'cifar10': self.n_class = 10
        elif dataset_name == 'cifar100': self.n_class = 100
        else: self.n_class = -1 # error

        if self.n_class != -1:
            if self.validation:
                self.dataloader, self.test_dataloader, _ = get_train_valid_loader(dataset_name, batch_size=self.batchsize,
                                                                                  augment=True, num_workers=1)
            else:
                self.dataloader, _, self.test_dataloader = get_train_valid_loader(dataset_name, batch_size=self.batchsize,
                                                                                  augment=True, num_workers=1)

            print('train num    ', self.dataloader.cardinality())
        else:
            print('\tInvalid input dataset name at CNN_train()')
            exit(1)
```

**b)**

Figura 34 - Código para executar o carregamento para cada dataset

O método executado quando um objeto da classe é chamado, está definido na Figura 35 a), estando esta cortada por motivos de simplicidade. Pode ser observado que, na implementação original, este método apenas executa uma rotina de treino, utilizando o *dataset* carregado e a rede neuronal obtida a partir do indivíduo.

Já na Figura 35 b), está a implementação modificada, onde é convertido o indivíduo para uma rede neuronal e o modelo é compilado com a mesma função de custo. Mesmo não sendo esta utilizada, é necessário para construir o modelo final. Após a construção da rede neuronal, é verificado o nome da métrica para assim passar um lote de imagens válido. Obtém-se assim o valor de fitness do indivíduo, sendo esta parte do código similar à do *Fast-Denser++*.

As funções referentes às métricas não serão mostradas novamente, pois a implementação é igual à apresentada no código modificado do *Fast-Denser++*, explicado anteriormente.

```python
def __call__(self, cgp, gpuID, epoch_num=200, out_model='mymodel.model'):
    if self.verbose:
        print('GPUID     :', gpuID)
        print('epoch_num :', epoch_num)
        print('batch_size:', self.batchsize)

    # model
    torch.backends.cudnn.benchmark = True
    model = CGP2CNN(cgp, self.channel, self.n_class, self.imgSize)
    init_weights(model, 'kaiming')
    model.cuda(gpuID)
    # Loss and Optimizer
    criterion = nn.CrossEntropyLoss()
    criterion.cuda(gpuID)
    optimizer = optim.Adam(model.parameters(), lr=0.01, betas=(0.5, 0.999))
    # optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, dampening=0, weight_decay=0.0005)
    input = torch.FloatTensor(self.batchsize, self.channel, self.imgSize, self.imgSize)
    input = input.cuda(gpuID)
    label = torch.LongTensor(self.batchsize)
    label = label.cuda(gpuID)

    # Train loop
    for epoch in range(1, epoch_num+1):
        start_time = time.time()
        if self.verbose:
            print('epoch', epoch)
        train_loss = 0
        total = 0
        correct = 0
        ite = 0
        for module in model.children():
            module.train(True)
        for _, (data, target) in enumerate(self.dataloader):
            if self.dataset_name == 'mnist':
                data = data[:,0:1,:,:] # for gray scale images
            data = data.cuda(gpuID)
```

**a)**

```python
def __call__(self, cgp, gpuID, epoch_num=200, out_model='mymodel.model'):
    if self.verbose:
        print('GPUID     :', gpuID)
        print('epoch_num :', epoch_num)
        print('batch_size:', self.batchsize)

    # model
    model = cgp2cnn(cgp, self.channel, self.n_class, self.imgSize)
    model.build((None, self.imgSize, self.imgSize, self.channel))

    # Loss and Optimizer
    model.compile(
        optimizer=tf.keras.optimizers.Adam(0.01, 0.5, 0.999),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )

    if self.eval_fn.__name__ == "relu_determinant":
        model(next(iter(self.dataloader))[0])

        # Passing only a batch of data to evaluate
        K_mat = self.eval_fn(model, next(iter(self.dataloader)))
        _, det = tf.linalg.slogdet(K_mat)
        t_acc = float(tf.get_static_value(det))

    elif self.eval_fn.__name__ == "ntk":
        # get only 32 batches for performance reasons
        batch = next(iter(self.dataloader))[0]
        x1, x2 = batch[:32,:,:,:], batch[32:64,:,:,:]
        eig_val_ratio = self.eval_fn(model, x1, x2)
        t_acc = -float(tf.get_static_value(eig_val_ratio))

    return t_acc
```

**b)**

Figura 35 - Código que avalia uma rede neuronal no CGP-CNN

### Preparação dos *datasets*

Na Figura 34 b), mostrou-se como são carregados os *datasets* através da função *get_train_valid_loader*, mas não se apresentou o código referente a essa função. Assim, na Figura 36 está a implementação totalmente refeita, pois a original utilizava *PyTorch*, enquanto agora utiliza o módulo *Tensorflow Datasets*.

Cada *dataset* neste módulo já tem *splits* próprios, ou seja, o CIFAR10 e 100 já têm o *split* de treino com 50 mil imagens selecionadas e outro *split* teste com 10 mil imagens, totalizando as 60 mil imagens de cada um destes *datasets*. Porém, como é utilizado ao longo do código um *dataset* de validação extraído do conjunto de dados de treino, calculou-se o número de imagens para treino e validação utilizando o parâmetro de *valid_size*. Obteve-se assim a percentagem de imagens que se deve retirar de cada *split* (conjunto de dados).

O *Tensorflow Datasets* permite receber uma lista explicitando o *split* de onde os dados são retirados e a percentagem de dados retirados. Por exemplo: *train[:90%]* – significa 90% de imagens, já: *train[90%:]* – significa retirar 10% de imagens, pois saltam-se os primeiros 90% do *split*.

Após obter os *datasets* de treino, validação e teste, normaliza-se as imagens para utilizarem valores de pixéis de 0 a 1 em vez de 0 a 255, além de criar as transformações para *data augmentation,* caso se deseje. Por fim, aplicam-se técnicas de *cache* e *prefetch* para garantir eficiência no carregamento e fornecimento dos dados ao longo das avaliações das redes neuronais e garante-se que os dados de treino são devidamente baralhados para prevenir sequências que prejudicam a aprendizagem. Os lotes de imagens também são construídos neste passo, onde são eliminados lotes que não tenham o número de imagens necessário, sendo neste caso 128. Esta eliminação é feita devido à métrica NTK, mas, novamente, não prejudica em nada a avaliação, pois todas as métricas utilizam apenas um pequeno conjunto de imagens (64 ou 128 imagens por métrica).

```python
def get_train_valid_loader(dataset_name,
                           batch_size,
                           augment,
                           valid_size=0.1,
                           show_sample=False,
                           num_workers=4):

    error_msg = "[!] valid_size should be in the range [0, 1]."
    assert ((valid_size >= 0) and (valid_size <= 1)), error_msg

    train_split = 100 - int(valid_size * 100)

    # Define splits for different datasets
    if dataset_name == 'cifar10' or dataset_name == 'cifar100':
        splits = [f'train[:{train_split}%]', f'train[{train_split}%:]', 'test']
    elif dataset_name == 'imagenet_resized':
        splits = [f'train[:{train_split}%]', f'train[{train_split}%:]', 'validation']
    else:
        print("Invalid Dataset!")
        exit(1)

    (train_ds, val_ds, test_ds), info = tfds.load(
        dataset_name,
        split=splits,
        with_info=True,
        as_supervised=True
    )

    normalize = tf.keras.layers.Rescaling(1./255)
    train_ds = train_ds.map(lambda x,y: (normalize(x), y), num_parallel_calls=num_workers)
    val_ds = val_ds.map(lambda x,y: (normalize(x), y), num_parallel_calls=num_workers)
    test_ds = test_ds.map(lambda x,y: (normalize(x), y), num_parallel_calls=num_workers)
```

```python
if augment:
    transform = tf.keras.Sequential([
        tf.keras.layers.RandomFlip("horizontal"),
        tf.keras.layers.Resizing(32, 32, crop_to_aspect_ratio=(4,4))
    ])
    train_ds = train_ds.map(lambda x,y: (transform(x), y), num_parallel_calls=num_workers)


train_ds = train_ds.cache().shuffle(buffer_size=int(info.splits['train'].num_examples * (1-valid_size)))
train_ds = train_ds.batch(batch_size, drop_remainder=True).prefetch(tf.data.AUTOTUNE)

val_ds = val_ds.cache().batch(batch_size, drop_remainder=True).prefetch(tf.data.AUTOTUNE)
test_ds = test_ds.cache().batch(batch_size, drop_remainder=True).prefetch(tf.data.AUTOTUNE)


# visualize some images
if show_sample:
    data_iter = next(iter(train_ds))
    images, labels = data_iter
    plot_images(images.numpy(), labels)

return (train_ds, val_ds, test_ds)
```

Figura 36 - Código para descarregar e transformar um dataset

### Descodificação genótipo / fenótipo

Na Figura 37 tem-se a versão modificada da função *cgp2cnn*, sendo esta responsável pela conversão do indivíduo num modelo *Keras*, ou seja, numa rede neuronal. Para tal, percorre-se a lista com as camadas/blocos que constituem o indivíduo e as suas conexões.

Com base no nome de cada nó (por exemplo, ConvBlock_128_3), é criada uma instância com a classe apropriada ao nó, tendo em atenção, caso seja um bloco convolucional ou residual, extrai-se o número e tamanho do *kernels* a partir do nome, considerando o exemplo anterior, seriam 128 *kernels* com tamanho 3x3. Por fim, percorre-se a lista de instâncias criadas anteriormente e faz-se a conexão com o(s) *input(s)* fornecidos na lista inicial de camadas e armazena-se o *output* noutra lista, pois estes *outputs* serão os *inputs* de outras camadas. Apenas o *Concat* e o *Sum* recebem dois *inputs*, pois um soma dois *inputs*, o outro *concatena* a dimensão de profundidade (canais) dos dois *inputs*.

```python
def cgp2cnn(cgp, in_channel, n_class, imgSize):
    layer_names = []
    encode = []
    n_class = n_class

    for name, in1, in2 in cgp:
        if name == 'input' in name:
            layer_names.append('input')
            encode.append(tf.keras.Input(shape=(imgSize, imgSize, in_channel)))
        elif name == 'full':
            layer_names.append('full')
            encode.append(tf.keras.layers.Dense(n_class))
        elif "Max" in name:
            layer_names.append('Max')
            encode.append(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
        elif "Avg" in name:
            layer_names.append('Avg')
            encode.append(tf.keras.layers.AveragePooling2D(pool_size=(2, 2)))
        elif name == 'Concat':
            layer_names.append('Concat')
            encode.append(Concat())
        elif name == 'Sum':
            layer_names.append('Sum')
            encode.append(Sum())
        else:
            _, func, out_size, kernel = name.split('_')
            out_size = int(out_size)
            kernel = int(kernel)
            if func == 'ConvBlock':
                layer_names.append('ConvBlock')
                encode.append(ConvBlock(out_size, kernel))
            else:
                layer_names.append('ResBlock')
                encode.append(ResBlock(out_size, kernel))

    outputs = [None] * len(cgp)

    for i, layer in enumerate(encode):
        if layer_names[i] == 'input':
            outputs[i] = layer
        elif layer_names[i] == 'Sum' or layer_names[i] == 'Concat':
            outputs[i] = layer(outputs[cgp[i][1]], outputs[cgp[i][2]])
        elif layer_names[i] == 'full':
            tmp = tf.keras.layers.Flatten()(outputs[cgp[i][1]])
            outputs[i] = layer(tmp)
        else:
            outputs[i] = layer(outputs[cgp[i][1]])

    return tf.keras.Model(inputs=outputs[0],outputs=outputs[-1])
```

Figura 37 - Código que converter genótipo num modelo *Keras*

Para implementar cada uma das camadas e blocos necessários para executar o CGP-CNN, apresentamos as funções na Figura 38. Como foi dito anteriormente, é necessário primeiro instanciar as classes. Porém, para simplificar e evitar erros inesperados no *Tensorflow*, decidiu-se utilizar funções para esta implementação (diferente da implementação original em *PyTorch*, onde realmente de utilizam classes). No entanto, para simular a instanciação dos objetos, criou-se uma função que retorna outra função interna, denominada de *block*. Assim, na "instanciação", é retornada a referência para essa função interna e quando o objeto é realmente chamado passando as conexões como argumento, na realidade, é chamada a respetiva função *block*.

A função *ConvBlock* é bastante autoexplicativa, sendo apenas necessário explicar porque se utiliza uma camada explícita de ativação em vez de se utilizar o argumento *activation* presente na camada *Conv2D*. Isto deve-se à implementação dos autores, a qual opta por em primeiro normalizar as saídas da convolução e só depois aplicar a função de ativação como explicado no artigo [77]. A ativação já terá os valores na sua distribuição probabilística, logo não faz sentido aplicar *Batch Normalization* após uma ativação. No entanto, o *Tensorflow*, por omissão, faz estas operações na ordem contrária, logo foi necessário fazer a ativação separadamente.

Na função *ResBlock* é necessário garantir que a dimensão de profundidade é igual entre o *input* e a saída, pois elas serão somadas no fim. Na implementação original, era criado um tensor de zeros com um tamanho na dimensão de profundidade igual à diferença entre as profundidades do i*nput* e saída. Depois concatenava-se no *input* ou na saída, dependendo qual tivesse maior dimensão, este tensor de zeros. No entanto, o *Tensorflow* e o *Keras* utilizam grafos de computação estáticos e dinâmicos, respetivamente. Logo, ao criar um tensor de zeros com o *Tensorflow* e ao criar camadas com o *Keras*, há conflitos devido à diferença no cálculo de dimensões, devido aos grafos serem dinâmicos e estáticos.

Para evitar este problema, utilizou-se outra técnica que garante a dimensão igual entre os dois tensores de i*nput* e saída. Esta consiste em aplicar uma convolução sem ativação e com *kernels* de tamanho 1x1, tendo o número de *kernels* desejados. Na função *Sum* também foi utilizada esta técnica, pois ambos os tensores têm de ter todas as dimensões iguais para serem somados. Para garantir que as outras dimensões também são iguais, o *Sum* e o *Concat* aplicam *Maximum Poolings* ao tensor com maior dimensão, diminuindo assim a sua dimensionalidade até coincidir com a do outro tensor, sendo esta técnica também utilizada na implementação original.

Com estas adaptações de código foram possíveis obter múltiplas topologias de redes neuronais, consideradas como melhores indivíduos, podendo agora serem treinadas intensivamente e avaliadas, tal como aquelas obtidas no *Fast-Denser++*. seu real desempenho.

```python
def ConvBlock(filters, kernel_size):
    def block(x):
        x = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')(x)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Activation('relu')(x)
        return x
    return block

def ResBlock(filters, kernel_size):
    def block(x):
        residual = x
        x = tf.keras.layers.Conv2D(filters, kernel_size, padding='same')(x)
        # Adjust the number of filters in the residual connection if necessary
        if residual.shape[-1] != filters:
            residual = tf.keras.layers.Conv2D(filters, (1, 1), padding='same')(residual)
        x = tf.keras.layers.BatchNormalization()(x)
        x = tf.keras.layers.Activation('relu')(x)
        x = tf.keras.layers.Add()([residual, x])
        return x
    return block

def Sum():
    def block(x ,y):
        while x.shape[1] != y.shape[1]:
            shape1 = x.shape[1:]
            shape2 = y.shape[1:]
            if shape1[0] > shape2[0]:
                x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
            elif shape1[0] < shape2[0]:
                y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(y)

        if x.shape[-1] < y.shape[-1]:
            x = tf.keras.layers.Conv2D(y.shape[-1], (1, 1), padding='same')(x)
        elif y.shape[-1] < x.shape[-1]:
            y = tf.keras.layers.Conv2D(x.shape[-1], (1, 1), padding='same')(y)

        return tf.keras.layers.Add()([x, y])
    return block

def Concat():
    def block(x ,y):
        while x.shape[1] != y.shape[1]:
            shape1 = x.shape[1:]
            shape2 = y.shape[1:]
            if shape1[0] > shape2[0]:
                x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(x)
            elif shape1[0] < shape2[0]:
                y = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(y)

        return tf.keras.layers.Concatenate()([x, y])
    return block
```

Figura 38 - Funções para criar camadas personalizadas do CGP-CNN

Tendo sido implementadas as versões modificadas de ambos os algoritmos, de seguida devem-se executar os algoritmos com ambas as métricas e com outras alterações que visem provar algo, além de treinar intensivamente as topologias obtidas para se poder observar a sua real exatidão e desempenho. O código e os resultados destes testes serão apresentados no seguinte capítulo.

# 6. Resultados da implementação

Para testar cada algoritmo com cada uma das métricas escolhidas, decidiu-se utilizar dois *datasets* que mostraram ser bastante comuns nos *benchmarks* dos mais variados algoritmos para procura de arquiteturas de redes neuronais convolucionais, sendo estes o CIFAR-10 e o CIFAR-100. Desta forma, o capítulo será dividido em duas partes: resultados do CIFAR-10 e resultados do CIFAR-100. É importante ter em atenção que tanto a procura como o treino das topologias encontradas foram executados em apenas uma GPU NVidia Tesla P100.

## 6.1. CIFAR-10

Executou-se assim o algoritmo *Fast-Denser++*, no *dataset* CIFAR10, três vezes: uma com o NTK, uma com o NASWOT, ambas utilizando o ficheiro de configuração e gramática original. A terceira execução consistiu numa repetição do NASWOT, sendo que a gramática apenas utilizou funções de ativação ReLU, já que, originalmente, havia mais possibilidades de funções de ativação a ser escolhidas.

Na Figura 39, estão as topologias obtidas, destacando-se a simplicidade e incorreta formação da topologia obtida pela métrica NTK( Figura 39 a). Isto porque, ter quatro camadas de *pooling* seguidas, não é algo comum e apenas uma convolução não é suficiente para aprender o *dataset*. Já a Figura 39 b), apresenta uma topologia obtida pelo NASWOT utilizando a configuração original. A topologia está razoavelmente construída, exceto as camadas *Dropout* entre convoluções, porém estas estão presentes na gramática original e decidiu-se mantê-las para manter os testes similares ao dos autores. Ter três camadas *Dropout* consecutivas também não é algo comum.

Estas particularidades das soluções afetaram significativamente os resultados obtidos após o treino. Por fim, a topologia na Figura 39 c), foi obtida pelo NASWOT com apenas funções de ativação ReLU. É a topologia mais profunda e tem a melhor construção entre as três. As duas últimas camadas de *Dropout* deveriam estar entre as duas camadas densas, no fim da rede, podendo reduzir o *overffiting* que foi sentido, posteriormente, ao longo do treino final. As imagens das topologias foram obtidas através do pacote *visualkeras*.

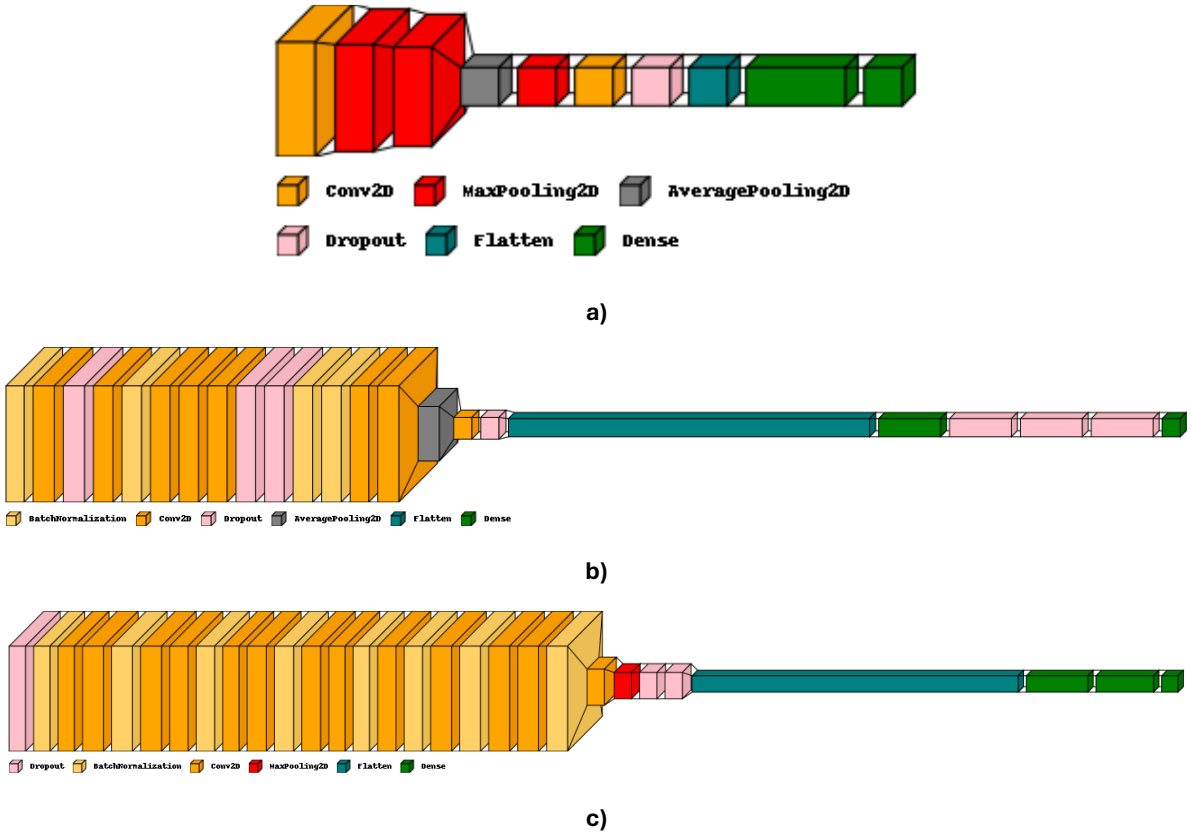


Figura 39 - Topologias encontradas

Para realizar o treino destas topologias, construiu-se um objeto *Sequential* utilizando o Keras (Figura 40), com as camadas e hiperparâmetros apresentados no fenótipo. Para treinar o modelo utilizam-se as configurações presentes nos artigos [78] e [71]. Utiliza-se assim, um tamanho de lote, *batch size*, de 128, 500 épocas de treino utilizando o optimizador SGD (*Stochastic Gradient Descent*) com um *momentum* de 0.9 e decaimento dos pesos de $5 \times 10^{-4}$, sendo que este decaimento funciona como um regularizador L2.

Também é necessário utilizar um *multistep learning rate scheduler*, presente na Figura 41 a), para garantir que o treino inicialmente dê mais importância à atualização dos pesos no início do que nas últimas épocas, fazendo assim um ajuste fino dos mesmos. Este *scheduler* foi ligeiramente alterado, considerando os artigos originais, pois após alguns testes, constatou-se que o *learning rate* de 0.1 é muito alto para o otimizador aplicar nos modelos obtidos. Obtiveram-se melhores resultados reduzindo por um fator de 10 todos os *learning rates*.

Também se utilizou um objeto *logger* para armazenar, ao longo do treino, as métricas obtidas em cada época, assim como um checkpoint que armazena e carrega os últimos pesos atualizados, guardando-os de 50 em 50 épocas, Figura 41 b). Desta forma, garante-se que, mesmo que o treino pare abruptamente, possa ser continuado posteriormente a partir do ponto de paragem.

```python
#Fast-Denser++ com ReLU Det e configs originais
model_name = "relu_det_original"
model = tf.keras.Sequential([
    tf.keras.Input(shape=(32,32,3)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(234, 5, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.Dropout(0.5686644689764315),
    tf.keras.layers.Conv2D(216, 2, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(216, 2, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.Conv2D(216, 2, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.Conv2D(216, 2, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.Dropout(0.7),
    tf.keras.layers.Dropout(0.48319400651119315),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Conv2D(205, 2, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.Conv2D(234, 5, padding="same", activation="relu", use_bias=False),
    tf.keras.layers.AveragePooling2D((4,4), strides=(2,2)),
    tf.keras.layers.Conv2D(131, 5, strides=2, padding="valid", activation="relu", use_bias=True),
    tf.keras.layers.Dropout(0.58861471336389),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(685, activation="relu", use_bias=False),
    tf.keras.layers.Dropout(0.58861471336389),
    tf.keras.layers.Dropout(0.3036107955292981),
    tf.keras.layers.Dropout(0.06695961697146902),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

Figura 40 - Exemplo de topologia de rede

```python
def scheduler(epoch, lr):
    if epoch < 5:
        return 0.001
    elif epoch >= 5 and epoch < 250:
        return 0.01
    elif epoch >= 250 and epoch < 375:
        return 0.001
    else:
        return 0.0001
```

**a)**

```python
model.compile(
    optimizer=tf.keras.optimizers.SGD(0.01, momentum=0.9, weight_decay=0.0005),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

model.summary()

batch_size = 128
epochs = 500
initial_epoch = 0

scheduler = tf.keras.callbacks.LearningRateScheduler(scheduler)
logger = tf.keras.callbacks.CSVLogger(model_name+'.csv', append=True, separator=';')

checkpoint_path = os.path.join(BASE_DIR, model_name+"/cp-{epoch:d}.weights.h5")
checkpoint_dir = os.path.dirname(checkpoint_path)

if not os.path.isdir(checkpoint_dir):
    os.makedirs(checkpoint_dir)

# Last weight
all_chkpt = os.listdir(checkpoint_dir)

if all_chkpt:
    chkpt_epochs = list( map(lambda x: int(x.split("-")[1].split(".")[0]), all_chkpt) )
    chkpt_epochs.sort(reverse=True)
    initial_epoch = chkpt_epochs[0]

    model.load_weights(checkpoint_path.format(epoch=initial_epoch))

# Save from 50 to 50 epochs
save_freq = 50 * (dataset['x_train'].shape[0]//batch_size)
checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                save_weights_only=True,
                                                save_freq=save_freq,
                                                verbose=1)
```

**b)**

Figura 41 - Configurações de treino

Ao criar-se os *callbacks* do *learning rate scheduler*, *logger* e *checkpoint*, estes podem ser passados para o método de treino do modelo, junto com o *dataset* obtido através de funções já implementadas no *Fast-Denser++*. A Figura 42 mostra o método *fit* que recebe os *callbacks* e o *dataset* CIFAR10, dividido em 50 mil imagens de treino e 6.5 mil de validação. No final, faz-se uma avaliação do modelo treinado com 3.5 mil imagens de teste, sendo esse o resultado colocado na Tabela 6.

```
score = model.fit(
    datagen_train.flow(dataset['x_train'],
                       dataset['y_train'],
                       batch_size=batch_size),
    epochs=epochs,
    validation_data=(datagen_test.flow(dataset['x_val'], dataset['y_val'], batch_size=batch_size)),
    callbacks=[scheduler, logger, checkpoint],
    initial_epoch=initial_epoch
)

model.evaluate(dataset['x_test'], dataset['y_test'], batch_size=batch_size)
```

Figura 42 - Treino do modelo com as configurações anteriores

Na Tabela 6 estão apresentados os resultados tanto da procura como do treino final. Pode-se constatar que a métrica que permitiu uma procura mais rápida foi a NTK, apesar de todas demorarem menos tempo que qualquer algoritmo apresentado no estado da arte anteriormente encontrado. As exatidões mostram-se insatisfatórias em relação ao modelo obtido no artigo original do *Fast-Denser++*, com 89,44% de exatidão, no CIFAR10, exceto para o algoritmo que utilizou NASWOT com a gramática original. Esse modelo foi obtido em 0.12 dias e completamente treinado em 0.55 dias, obtendo uma exatidão de 91%, ao contrário do algoritmo original que demorou 2.29 dias para obter 89,44% de exatidão.

Quando se utilizou o NASWOT apenas com funções de ativação ReLU, houve uma redução no espaço de procura, porém a exatidão e o tempo despendido foram piores. Isso deve-se ao NASWOT ser ganancioso, já que ignora quaisquer camadas que não tenham uma função de ativação ReLU. Como neste cenário só há esse tipo de funções ativação, ele avaliará muitas mais camadas, havendo maior probabilidade de haver diferenças entre as ativações, obtendo valores de *fitness* superiores. Assim, haverá muitas mais camadas com ReLU, incluindo camadas densas que tendem a sofrer de *overffiting*. Por esse motivo, demorou mais tempo a treinar e obteve piores resultados.

Tabela 6 - Resultados dos algoritmos no *Fast-Denser++* e dos treinos dos melhores candidatos no CIFAR10

| Algoritmo | Tempo de procura (em dias) | Tempo de treino (em dias) | Exatidão | Número de Parâmetros | Topologia | Observações |
|---|---|---|---|---|---|---|
| Fast-DENSER++ com NTK | 0.06 | 0.03 | 0.46 | 80 471 | Figura 39 a) | Utilizou as configurações dos autores |
| Fast-DENSER++ com NASWOT | 0.12 | 0.55 | 0.91 | 6 162 373 | Figura 39 b) | Utilizou as configurações dos autores |
| Fast-DENSER++ com NASWOT | 0.19 | 1.21 | 0.77 | 20 425 526 | Figura 39 c) | Utilizou as configurações dos autores, mas só há ReLUs |

Para complementar os resultados, obteve-se o gráfico da Figura 43, utilizando o pacote *Seaborn* e *Pandas* para plotar e carregar os ficheiros com as métricas de treino, obtidas época a época.. A métrica NTK realmente mostrou não aprender satisfatoriamente o *dataset* CIFAR10, mantendo sempre exatidões baixas, porém não mostrou qualquer tipo de *overffiting*.

Como já constatado anteriormente, apenas pela observação dos valores da Tabela 6, o algoritmo que utilizou a métrica de fitness NASWOT, utilizando uma gramática com apenas funções de ativação ReLU, apresentou um claro *overffiting* já que ao longo do treino obteve uma exatidão próxima de 100%, mas na validação não alcançou os 80%. Também apresentou elevada instabilidade, constatada pela linha da exatidão e custo de validação. O algoritmo que utilizou o NASWOT, mantendo a gramática original, não mostrou qualquer sinal de *overffiting*, além de apresentar uma aprendizagem praticamente perfeita, sendo assim um bom algoritmo candidato para se utilizado, pelo menos no *dataset* CIFAR10 ou *datasets* com características semelhantes.

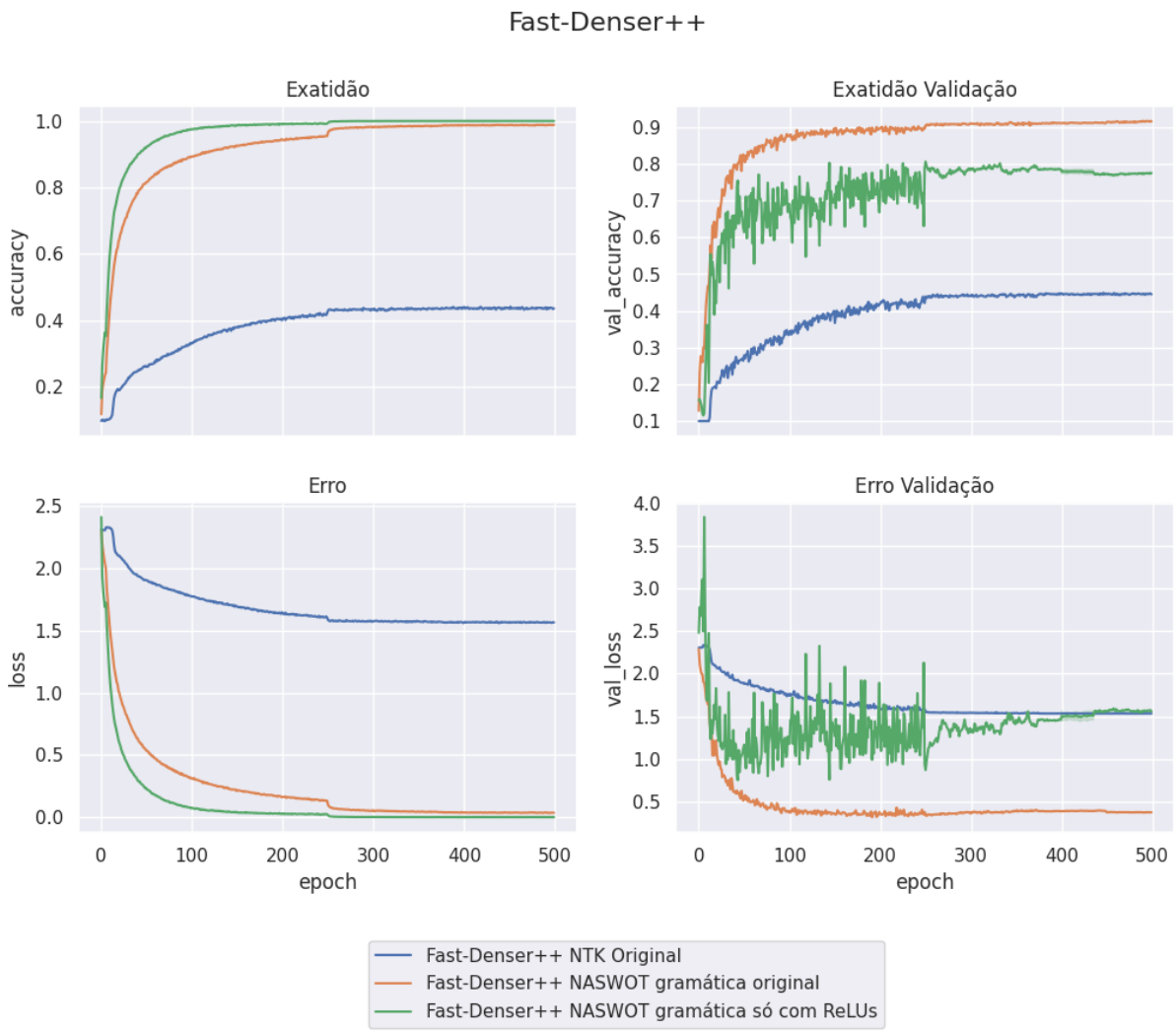


Figura 43 - Métricas de treino ao longo das épocas no *Fast-Denser++*

Os resultados no *Fast-Denser++*, em geral, mostraram-se satisfatórios, mesmo mantendo a liberdade e complexidade da gramática utilizada. No entanto, fazer uso de camadas em vez de blocos de camadas como elementos de construção da rede, aumenta exponencialmente o tamanho do espaço de procura, aumentando a complexidade do problema de otimização. Ao longo das procuras, foi observado que a procura usando a métrica NTK apresenta instabilidades nos resultados e tende a privilegiar redes com poucos pesos. Já a utilização de NASWOT, leva a uma procura gananciosa onde quanto maior o número de camadas com a função de ativação ReLU, maior tende a ser o seu resultado, logo melhor *fitness* o indivíduo tem.

Para treinar as topologias obtidas no CGP utilizou-se o mesmo *pipeline* do *Fast-Denser++*, alterando apenas a forma de criar o modelo, pois este não será um modelo sequencial. Adaptou-se assim a função *cgp2cnn*, para criar um modelo que no fim terá uma camada *Flatten* quer permite conectar o *output* do indivíduo (última camada da topologia) à camada densa que faz a classificação para as 10 possíveis classes do *dataset* CIFAR10.

Os resultados obtidos são apresentados na Tabela 7, onde se constata que o tempo de procura não só é igual entre as diferentes métricas, mas também é idêntico ao tempo gasto pelo *Fast-Denser++*. Assim, pode-se concluir que as métricas são invariantes entre estes dois algoritmos, relativamente ao tempo de procura. No tempo de treino, as redes neuronais obtidas através da métrica NTK foram significativamente mais rápidas, devido à simplicidade da sua topologia.

A utilização do ResSet, mostrou-se insatisfatória relativamente à exatidão obtida. Possivelmente, o *learning rate scheduler* está em conflito com estas topologias, como aconteceu inicialmente no *Fast-Denser++*, devendo ser ajustado. Porém, como têm sido obtidos bons resultados utilizando este *scheduler*, não se voltará a ajustar para evitar refazer testes e poder prejudicar outras combinações de algoritmos e métricas. A nível de parâmetros, o NASWOT obtém sempre mais parâmetros por ser ganancioso e o NTK produz sempre um número baixo de parâmetros por desenvolver topologias muito simples.

Tabela 7 – Resultados dos algoritmos no CGP-CNN e dos treinos dos melhores candidatos no CIFAR10

| Algoritmo | Tempo de procura (em dias) | Tempo de treino (em dias) | Exatidão | Número de Parâmetros | Topologia[1] | Observações |
|---|---|---|---|---|---|---|
| CGP-ConvSet com NASWOT | 0.1 | 0.3 | 0.92 | 2 784 426 | [79] | Algoritmo original |
| CGP-ConvSet com NTK | 0.11 | 0.06 | 0.85 | 329 354 | [80] | Algoritmo original |
| CGP-ResSet com NASWOT | 0.12 | 0.62 | 0.1 | 5 194 506 | [81] | Algoritmo original |
| CGP-ResSet com NTK | 0.11 | 0.07 | 0.1 | 894 922 | [82] | Algoritmo original |

Para complementar a análise dos resultados no CGP-CNN, criou-se a Figura 44, onde se pode verificar que o custo na utilização do *ResSet* não está a ser mostrado, por ser extremamente elevado. Destaca-se o excelente resultado do NASWOT utilizando o *ConvSet*, sendo próximo do NTK utilizando também o *ConvSet*. As linhas dos dois parecem sincronizadas em certos pontos, sendo isso causado pela mudança dos *learning rates*. Assim, no CGP-CNN, o melhor resultado, a nível de exatidão, foi o NASWOT *ConvSet*. No entanto, a nível de tempo, o NTK *ConvSet* demorou apenas 4,08 horas, enquanto o NASWOT *ConvSet* demorou 9,8 horas no total. Ao nível de procura demoraram ambos 2,4 horas. Considerando a instabilidade e os fracos resultados no *Fast-Denser++*, o NTK não será considerado o melhor resultado nestes testes envolvendo o CGP-CNN.

[1] Devido ao elevado número de camadas da topologia, foi necessário partilhar as imagens hospedadas externamente na Internet, pois não é viável apresentá-las neste documento

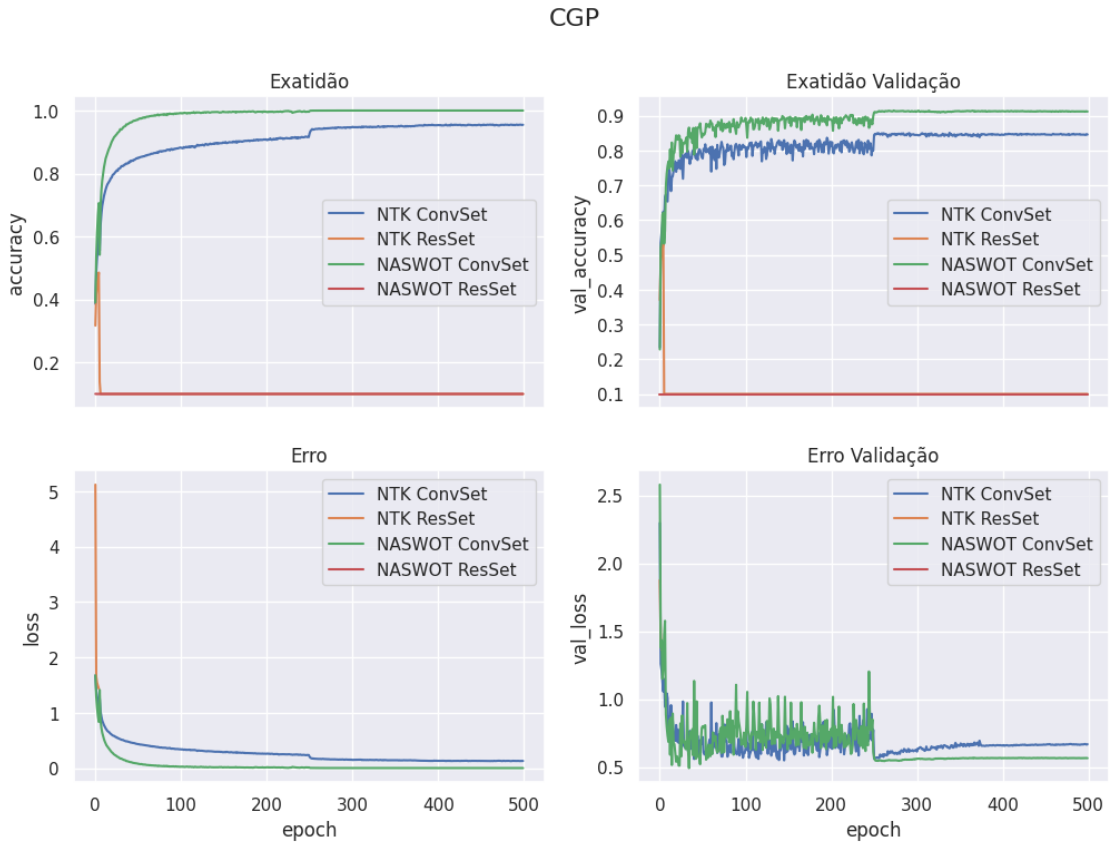


Figura 44 – Métricas de treino ao longo das épocas no CGP-CNN no CIFAR-10

Comparando os resultados entre CGP-CNN e *Fast-Denser++*, verifica-se que, apesar de tempos similares de procura, ainda há algumas diferenças. Enquanto no CGP-CNN os algoritmos demoraram 2,4 horas, no *Fast-Denser++* houve algoritmos a demorar o dobro. Ainda mais significativo foi o tempo de treino, sendo que o CGP-CNN demorou 9,8 horas, enquanto no *Fast-Denser++* houve algoritmos a demorarem mais de um dia a treinar a topologia selecionada. Isso deve-se ao facto de não haver camadas densas no espaço de procura do CGP-CNN, além de ele limitar mais esse espaço do que o *Fast-Denser++*. Um benefício dessa limitação, é que o NTK conseguiu melhores resultados a nível de exatidão e tempo, sendo que o tempo foi algo que melhorou em geral.

Na próxima fase, serão testados os algoritmos e métricas aplicadas no *dataset* CIFAR100. Este é idêntico ao CIFAR10, porém em vez de 10 classes de imagens possui 100 classes, aumentando o desafio das redes neuronais para obterem exatidões elevadas na classificação. Adicionalmente, este *dataset* contem um menor número de imagens por classe, tornando este desafio ainda mais difícil. Considerando o grau de dificuldade, serão apenas utilizados os dois melhores algoritmos encontrados até este momento, sendo eles o NASWOT *Fast-Denser++* com gramática original e o NASWOT *ConvSet* do CGP-CNN.

## 6.2. CIFAR-100

A Tabela 8 contém os resultados das variações dos dois melhores algoritmos com a métrica NASWOT. O *Fast-Denser++* mostrou-se incapaz de aprender o *dataset* e resultou na arquitetura com maior número de parâmetros, pois a sua gramática contém camadas densas. Estas as camadas que fazem crescer exponencialmente o número de parâmetros do modelo final. O CGP mostrou-se razoável, não se podendo comparar com os resultados dos algoritmos originais, pois os autores não efetuaram testes no CIFAR-100. No entanto, baseado no estado da arte obtido anteriormente, a exatidão ficou abaixo dos esperados 81%.

O algoritmo original com NASWOT do CGP apresentou um modelo com número significativo de parâmetros, cerca de 28 milhões. Considerando os 68% de exatidão obtidos, este resultado sugere um padrão de *overffiting*, já que muitos parâmetros tendem a fazer *overffit d*o modelo. Uma forma de diminuir o número de parâmetros é não utilizar a camada de *flatten*, que vetoriza todo o *output* do *feature extractor*, sendo esta substituída por uma camada *Global Average Pooling*, que resume o *output* do *feature extractor* em apenas 1 pixel.

Esta técnica foi proposta em [83] exatamente para evitar *overffiting*, mostrando resultados positivos. Assim, treinou-se a topologia com esta alteração diminuindo consideravelmente o número de parâmetros, para cerca de 1,8 milhões.Os resultados mantiveram-se, continuando assim a fazer *overffit do modelo*. Também se obteve uma nova topologia utilizando esta técnica, porém a mesma mostrou-se inferior à anterior, com apenas 63% de exatidão.

Outra técnica utilizada para tentar minimizar o *overffiting* consiste em colocar algumas camadas densas após a camada de *flatten* para que a rede possa aprender com o *output* do *feature extractor* em vez de tentar logo classificar. Colocou-se assim uma camada densa com 512 unidades e um *dropout* com 50% probabilidade de desconectar cada unidade, para prevenir *overffiting*. Aplicando-a no treino, a exatidão aumentou para 71%, porém o número de parâmetros aumentou 100 vezes, ficando o modelo com 136 milhões de parâmetros.

A nível de tempo, todas as experiências demoraram 0,1 dias (2,4 horas) a procurar a topologia e cerca de 0,4 dias (9,6 horas) no seu treino. Novamente, o tempo de procura não só superou significativamente o tempo dos algoritmos originais, como também superou os 0,2 dias do estado da arte. A nível de treino, os dois algoritmos que utilizaram mais camadas densas, demoraram mais tempo a serem treinados, com o *Fast-Denser++* a demorar 0,55 dias (13,2 horas).

Tabela 8 – Resultados dos algoritmos e dos treinos dos melhores candidatos no CIFAR100

| Algoritmo | Tempo de procura (em dias) | Tempo de treino (em dias) | Exatidão | Número de Parâmetros | Topologia[2] | Observações |
|---|---|---|---|---|---|---|
| *Fast-Denser++* com NASWOT | 0.13 | 0.55 | 1,5% | 101 945 718 | [84] | Algoritmo original |
| *CGP-ConvSet* com NASWOT | 0.10 | 0.42 | 68% | 27 988 580 | [85] | Algoritmo original |
| *CGP-ConvSet* com NASWOT | 0.10 | 0.40 | 63% | 2 219 140 | [86] | Algoritmo original utilizando *Global Average Pooling* e *pipeline* de treino com uma *GlobalAveragePooling* |
| *CGP-ConvSet* com NASWOT | 0.10 | 0.47 | 71% | 136 043 620 | [85] | Algoritmo original e *pipeline* de treino com uma camada densa e uma de *dropout* |
| *CGP-ConvSet* com NASWOT | 0.10 | 0.44 | 68% | 1 799 780 | [85] | Algoritmo original e *pipeline* de treino com uma *Global Average Pooling* |

Como a análise de todos estes resultados através de uma tabela não é fácil, gerou-se a Figura 45, onde é possível observar o modelo do *Fast-Denser++* a sofrer de elevado *underffiting*. A CGP original foi a pior entre todas a nível de erro de validação. A configuração que utiliza *dropouts* nos treinos mostrou os melhores resultados e as configurações que utilizaram *Global Average Pooling* no treino e/ou na procura tiveram resultados medianos. Assim, se o objetivo for obter um modelo leve, utilizar a topologia obtida pelo CGP NASWOT original, treinado com *Global Average Pooling* será a melhor opção.

[2] Devido ao elevado número de camadas da topologia, foi necessário partilhar as imagens hospedadas externamente na Internet, pois não é viável apresentá-las neste documento

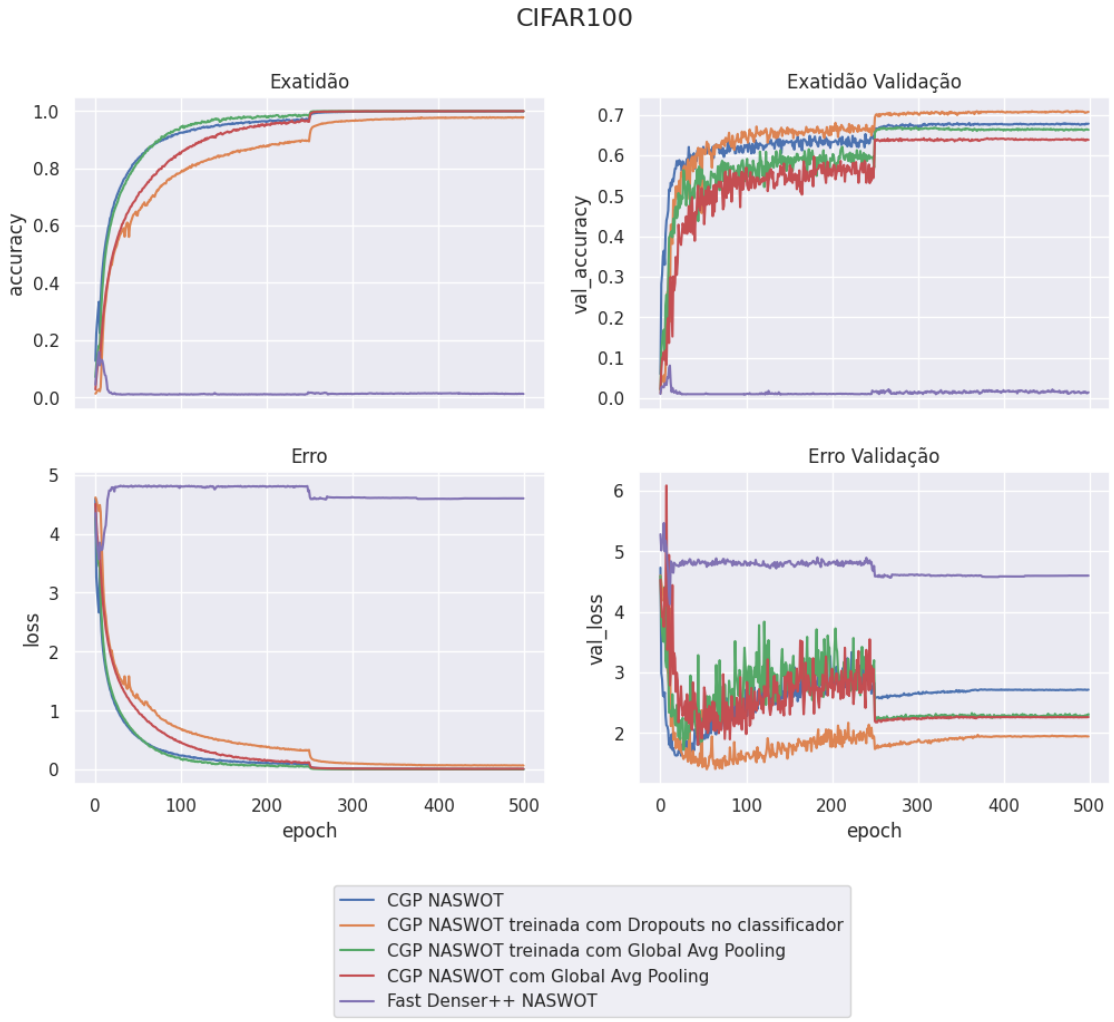


Figura 45 - Métricas de treino ao longo das épocas no CIFAR-100

Os resultados aqui apresentados em ambos os *datasets* mostraram-se razoáveis, tendo sido superado o estado da arte em tempo de procura e ficando próximo em relação à exatidão, considerando que se estão a utilizar algoritmos complexos que originalmente demoraram vários dias para fazer a procura. No capítulo seguinte realizar-se-á uma discussão e resumo de todos estes resultados.

# 7. Discussão dos resultados

No capítulo anterior foram apresentados todos os resultados obtidos após a adaptação dos algoritmos de programação genética para utilizarem as métricas identificadas como medida de desempenho das redes evoluídas. Os resultados, utilizando NASWOT, no *dataset* CIFAR-10 superaram os resultados do *Fast-Denser++* original. Consideramos que tal facto se deve ao algoritmo poder fazer uma procura muito mais rápida, considerando as estimativas da métrica NASWOT, podendo assim obter arquiteturas muito melhores.

A métrica NTK mostrou-se insatisfatória, não só no algoritmo *Fast-Denser++,* mas também no CGP-CNN. Um dos fatores que pode ter contribuído para o mau desempenho deve-se à falta de recursos sobre a implementação desta métrica, tendo-se encontrado apenas uma implementação parcial em *PyTorch.* Tanto o *Tensorflow* como o *JAX* (versão para computação numérica de elevada performance do *Tensorflow*) mostraram não conseguir lidar com grandes lotes de exemplos sem exceder a memória da GPU, obrigando à utilização de lotes pequenos, podendo assim prejudicar as estimativas.

O algoritmo *Fast-Denser++* também mostrou resultados melhores com a gramática original do que com a gramática modificada por utilizar apenas unidades *ReLUs*. Numa primeira análise, tal facto pode parecer incorreto, pois todas as camadas que não tenham função de ativação ReLU são ignoradas na estimativa do valor de *fitness* utilizando a métrica NASWOT. No entanto, isso pode servir de travão para esta métrica gananciosa, ao manter certas camadas que utilizarão outras funções de ativação que podem fazer estimativas erráticas. Tal funcionará como prevenção de *overffiting,* já que as camadas ReLU terão de equilibrar a rede.

Já o algoritmo CGP com os blocos convolucionais residuais (ResSet), mostrou resultados insatisfatórios, possivelmente devido ao *learning rate scheduler* utilizado, pois os valores poderiam ter de ser ajustados especificamente para este tipo de blocos convolucionais. No entanto, tal não foi feito para manter o *pipeline* treino semelhante para todos os algoritmos testados, garantindo uma comparação justa. Já o *ConvSet* com NASWOT mostrou resultados impressionantes. A topologia encontrada inclui conexões residuais entre camadas iniciais e camadas profundas da rede, ou seja, houve a passagem de informação inicial para o fim da arquitetura, garantindo que as entradas não são "esquecidas" podendo ser este o segredo para o excelente resultado de 92% de exatidão.

Quanto aos testes realizados utilizando o *dataset* CIFAR-100, reduziu-se o número de algoritmos, utilizando apenas os dois melhores da fase anterior, considerando que os algoritmos que obtiveram resultados insatisfatórios no CIFAR-10, dificilmente terão um bom desempenho no dataset CIFAR-100. O *Fast-Denser++* mostrou-se incapaz de o aprender, possivelmente, por ter demasiada liberdade na construção das arquiteturas e ter uma maior tendência a fazer *overffit* devido à inclusão de camadas densas.

Já o CGP atingiu um desempenho mais satisfatório, 68% de exatidão, porém ainda superado pelo estado da arte onde foram atingidos 81% de exatidão. Tentaram-se várias técnicas para aumentar a exatidão das redes evoluídas e diminuir o claro *overffiting*, concluindo-se que utilizar camadas densas no treino, colocando-as após a vectorização do *feature extractor*, aumentou ligeiramente a exatidão e exponencialmente o número de parâmetros.

A topologia obtida mostrou-se bastante complexa, seguindo, porém, a mesma lógica da topologia encontrada no CIFAR-10 pelo CGP. Nomeadamente, pode-se observar a utilização de múltiplas conexões residuais ao longo da rede para garantir que informações passadas estejam presentes nas camadas mais profundas. Ainda assim não foi o suficiente para obter uma topologia que supere o estado da arte. O problema pode estar na estratégia do algoritmo genético. Seria plausível alterar a estratégia para que não houvesse apenas um pai e sim uma população completa que permitisse recombinação e mutação para obter-se uma maior diversidade de indivíduos. No entanto esta estratégia resultaria numa maior complexidade computacional do algoritmo.

Considerando que a utilização de NASWOT como métrica de avaliação mostrou ser superior quando comparada com o desempenho dos algoritmos originais que avaliavam as soluções através do treino efetivo das redes neuronais, podemos concluir que efetivamente esta métrica produz valores diretamente proporcionais à exatidão de uma rede neuronal evoluída por um algoritmo que a empregue para avaliar as soluções.

A pergunta sobre se um algoritmo *surrogate* pode ter melhores resultados que estas métricas, também foi respondida, pois se o próprio algoritmo original, treinando tradicionalmente os indivíduos, obteve piores resultados do que os obtidos utilizando NASWOT, então um algoritmo *surrogate* que treina parcialmente os indivíduos e utiliza estimativas em paralelo, seria no máximo igual a nível de exatidão e inferior a nível de tempo de procura, i.e., necessitaria de mais tempo de procura.

Os *datasets* CIFAR-10 e CIFAR-100 mostraram resultados distintos, apesar de terem ambos 60 mil imagens divididas, originalmente, em 50 mil para treino e 10 mil para teste, com um tamanho 32x32. Porém, enquanto o CIFAR-10 tem apenas 10 classes e 6 mil imagens por classe, o CIFAR-100 tem 100 classes e 600 imagens por classe. Implicando um desafio muito superior, o qual o *Fast-Denser++* com NASWOT não conseguiu superar. O CGP-CNN com NASWOT conseguiu um resultado razoável, mas é visível que o *feature exctractor* sozinho não é suficiente, sendo necessárias camadas densas e técnicas para prevenir *overffiting* como a adição de camadas *dropout*.

Outro aspeto importante relaciona-se com o tamanho das imagens, já que estas são bastante pequenas, sendo complicado identificar detalhes suficientes que separem devidamente cada classe de imagens. Esse problema não foi identificado durante a utilização do CIFAR-10, talvez por haver mais imagens por classe, porém no CIFAR-100, não só causou *overffiting*, pois não permitiu que o modelo aprendesse a

generalizar a suas classificações, como também provocou *underffiting* no *Fast-Denser++*, impossibilitando-o de aprender os detalhes das imagens. Abre-se assim uma nova avenida de investigação, na qual se poderá estudar a utilização de outros algoritmos de programação genética, possivelmente utilizando DAGs, porém optando pelo uso de outras estratégias evolutivas para tentar melhorar a exatidão das topologias finais. O facto de ter sido obtido um tempo de procura de apenas 2.4 horas, em ambos os *datasets*, utilizando apenas uma GPU permite que esta abordagem seja viável para a procura de arquiteturas de redes neuronais com baixos recursos computacionais.

# 8. Conclusão

No desenvolvimento da primeira fase deste projeto, foi possível rever a literatura relevante face ao problema da procura de arquiteturas para redes neuronais convolucionais. Algoritmos genéticos e de inteligência de enxame foram os algoritmos encontrados mais frequentemente, devido à simplicidade das suas representações. Contudo, o PSO mostrou-se o tipo de algoritmo mais promissor após observação dos resultados apresentados na literatura. Esta análise possibilitou ainda identificação de técnicas aplicadas a estes algoritmos que permitem minimizar o tempo de procura

Aquelas técnicas incluem a avaliação das arquiteturas através de modelos *surrogate* ou metodologias que substituem por completo o treino das redes neuronais, diminuindo assim significativamente o tempo de procura e mantendo resultados razoáveis em termos de exatidão das soluções finais. Outras técnicas, como a utilização de blocos de camadas em vez de camadas únicas, assim como a partilha de pesos entre diferentes *kernels*, também contribuiu para a diminuição do tempo, embora limitando o espaço de procura.

A utilização de um *autoencoder* que não utiliza retro-propagação para aprender a distribuição de um conjunto de imagens, aplicando assim uma redução de dimensionalidade nas entradas das redes, mostrou-se incapaz de extrair a informação relevante, de forma que a rede mantivesse os resultados e convergisse em menos tempo.

No fim desta análise, mantiveram-se algumas questões em aberto, como por exemplo, se a avaliação por treino das redes for substituída por uma das técnicas de avaliação identificadas, utilizando, porém, algoritmos com representações mais complexas como (árvores, grafos, gramáticas etc.), que normalmente obtêm melhores resultados, mas exigem mais recursos computacionais, será que os resultados se mantêm ou diminuirão significativamente? Outra questão relevante é se modelos *surrogate* tendem a ter menos erros de avaliação do que os algoritmos cuja avaliação da rede não utiliza qualquer tipo de treino.

Considerando essas questões em aberto, decidiu-se, na segunda parte deste projeto, explorar a utilização de alguns algoritmos promissores com representações mais complexas do que aquelas encontradas no estado da arte. Estes são algoritmos que tendem a demorar vários dias para encontrar a melhor topologia, sendo assim candidatos adequados para testar a utilização de métricas, que farão a estimativa das exatidões sem qualquer tipo de treino, mais especificamente as métricas NASWOT e NTK.

Foi necessário adaptar as implementações dos dois algoritmos de programação genética para que utilizassem as métricas referidas, tendo sido estas também implementadas em *Tensorflow*. Os resultados mostraram-se satisfatórios tanto no CIFAR-10 como no CIFAR-100, não superando os resultados do estado da arte a nível de exatidão, mas ultrapassando-os a nível de tempo de procura, mostrando que o

NASWOT é uma métrica invariável em relação ao *dataset*, no que diz respeito aos datasets considerados neste projeto

Concluiu-se assim que o algoritmo CGP-CNN utilizando a métrica NASWOT seria a melhor combinação tendo em conta os dois algoritmos de programação genética selecionados, constituindo uma abordagem viável à procura automática de arquiteturas de redes neuronais utilizando baixos recursos computacionais. No entanto, ainda há espaço para melhoria, pois apesar de a representação baseada em DAG proporcionar conexões livres e uma representação interpretativa e fiel a uma rede neuronal, a estratégia de evolução utilizada no CGP-CNN, pode ser melhorada para abranger indivíduos diversos, na esperança de se conseguir superar as exatidões do estado da arte nestes dois *datasets*.

# Referências bibliográficas


[1] Y. LeCun, «Generalization and network design strategies», *Connectionism in perspective*, Jun. 1989, Acedido: 12 de Janeiro de 2024. [Em linha]. Disponível em: https://www.academia.edu/2813343/Generalization_and_network_design_strategies

[2] A. Krizhevsky, I. Sutskever, e G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks», Acedido: 20 de Janeiro de 2024. [Em linha]. Disponível em: http://code.google.com/p/cuda-convnet/

[3] K. Simonyan e A. Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition», *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, Set. 2014, Acedido: 1 de Janeiro de 2024. [Em linha]. Disponível em: https://arxiv.org/abs/1409.1556v6

[4] C. Szegedy *et al.*, «Going Deeper with Convolutions», *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June-2015, pp. 1–9, Set. 2014, doi: 10.1109/CVPR.2015.7298594.

[5] K. He, X. Zhang, S. Ren, e J. Sun, «Deep Residual Learning for Image Recognition», *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dez. 2015, doi: 10.1109/CVPR.2016.90.

[6] G. Huang, Z. Liu, L. Van Der Maaten, e K. Q. Weinberger, «Densely Connected Convolutional Networks», *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 2261–2269, Ago. 2016, doi: 10.1109/CVPR.2017.243.

[7] B. Zoph e Q. V. Le, «Neural Architecture Search with Reinforcement Learning», *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Nov. 2016, Acedido: 20 de Janeiro de 2024. [Em linha]. Disponível em: https://arxiv.org/abs/1611.01578v2

[8] Z. H. Zhan, J. Y. Li, e J. Zhang, «Evolutionary deep learning: A survey», *Neurocomputing*, vol. 483, pp. 42–58, Abr. 2022, doi: 10.1016/J.NEUCOM.2022.01.099.

[9] B. Wang, B. Xue, e M. Zhang, «Surrogate-Assisted Particle Swarm Optimization for Evolving Variable-Length Transferable Blocks for Image Classification», *IEEE Trans Neural Netw Learn Syst*, vol. 33, n. 8, pp. 3727–3740, Ago. 2022, doi: 10.1109/TNNLS.2021.3054400.

[10] J. Huang, B. Xue, Y. Sun, e M. Zhang, «Multi-Objective Evolutionary Search of Compact Convolutional Neural Networks with Training-Free

Estimation», em *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, New York, NY, USA: ACM, Jul. 2023, pp. 655–658. doi: 10.1145/3583133.3590535.

[11] J. Mellor, J. Turner, A. Storkey, e E. J. Crowley, «Neural Architecture Search without Training», 2021, Acedido: 3 de Janeiro de 2024. [Em linha]. Disponível em: https://github.com/

[12] S. Russel e P. Norvig, «Artificial intelligence—a modern approach 3rd Edition», *Knowl Eng Rev*, 2012, doi: 10.1017/S0269888900007724.

[13] W. S. McCulloch e W. Pitts, «A logical calculus of the ideas immanent in nervous activity», *Bull Math Biophys*, vol. 5, n. 4, pp. 115–133, Dez. 1943, doi: 10.1007/BF02478259/METRICS.

[14] D. 0 Hebb, «The Organization of Behavior A NEUROPSYCHOLOGICAL THEORY».

[15] C. Van Der Malsburg, «Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms», *Brain Theory*, pp. 245–248, 1986, doi: 10.1007/978-3-642-70911-1_20.

[16] R. M. Friedberg, «A Learning Machine: Part I», *IBM J Res Dev*, vol. 2, n. 1, pp. 2–13, Abr. 2010, doi: 10.1147/RD.21.0002.

[17] R. M. Friedberg, B. Dunham, e J. H. North, «A Learning Machine: Part II», *IBM J Res Dev*, vol. 3, n. 3, pp. 282–287, Abr. 2010, doi: 10.1147/RD.33.0282.

[18] J. H. (John H. Holland, «Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence», p. 183, 1975.

[19] «Deep Learning». Acedido: 30 de Dezembro de 2023. [Em linha]. Disponível em: https://www.deeplearningbook.org/

[20] D. C. Plaut, S. J. Nowlan, G. E. Hinton, e S. Nowlan, «Experiments on Learning by Back Propagation», 1986.

[21] S. Raschka e V. Mirjalili, «Python machine learning : machine learning and deep learning with python, scikit-learn, and tensorflow 2», pp. 20–51.

[22] A. L. Maas, A. Y. Hannun, e A. Y. Ng, «Rectifier Nonlinearities Improve Neural Network Acoustic Models», 2013.

[23] Ž. Đ. Vujović, «Classification Model Evaluation Metrics», *IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 12, n. 6, p. 2021, Acedido: 16 de Janeiro de 2024. [Em linha]. Disponível em: www.ijacsa.thesai.org

[24] «An Introduction to Genetic Algorithms». Acedido: 12 de Janeiro de 2024. [Em linha]. Disponível em:

https://mitpress.mit.edu/9780262631853/an-introduction-to-genetic-algorithms/

[25] M. A. Albadr, S. Tiun, M. Ayob, e F. Al-Dhief, «Genetic Algorithm Based on Natural Selection Theory for Optimization Problems», *Symmetry 2020, Vol. 12, Page 1758*, vol. 12, n. 11, p. 1758, Out. 2020, doi: 10.3390/SYM12111758.

[26] S. Katoch, S. S. Chauhan, e V. Kumar, «A review on genetic algorithm: past, present, and future», *Multimed Tools Appl*, vol. 80, n. 5, p. 8091, Fev. 2021, doi: 10.1007/S11042-020-10139-6.

[27] L. M. Hiot *et al.*, «Handbook of Swarm Intelligence», *Springer*, vol. 8, pp. 221-239–239, 2010, Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: http://www.springerlink.com/content/r4k7w76467163180/

[28] G. Beni, J. Wang, e S. Hackwood, «Swarm Intelligence in Cellular Robotic Systems», *NATO ASI Series*, vol. 102, pp. 703–712, 1989, Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: http://dx.doi.org/10.1007/978-3-642-58069-7_38

[29] H. KAWAMURA, M. YAMAMOTO, e A. OHUCHI, «Positive Feedback as a Search Strategy», *Technical Report*, vol. 37, n. 5, pp. 91–16, 1991, doi: 10.9746/SICETR1965.37.455.

[30] Y. Wan, Y. Zhong, A. Ma, e L. Zhang, «An Accurate UAV 3-D Path Planning Method for Disaster Emergency Response Based on an Improved Multiobjective Swarm Intelligence Algorithm», *IEEE Trans Cybern*, vol. 53, n. 4, pp. 2658–2671, Abr. 2023, doi: 10.1109/TCYB.2022.3170580.

[31] J. Kennedy e R. Eberhart, «Particle swarm optimization», *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948, doi: 10.1109/ICNN.1995.488968.

[32] M. J. Page *et al.*, «The PRISMA 2020 statement: an updated guideline for reporting systematic reviews», *BMJ*, vol. 372, Mar. 2021, doi: 10.1136/BMJ.N71.

[33] «Mendeley». Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: https://www.mendeley.com/

[34] «ACM Digital Library». Acedido: 1 de Janeiro de 2024. [Em linha]. Disponível em: https://dl.acm.org/

[35] «IEEE Xplore». Acedido: 1 de Janeiro de 2024. [Em linha]. Disponível em: https://ieeexplore.ieee.org/Xplore/home.jsp

[36] «Scopus». Acedido: 1 de Janeiro de 2024. [Em linha]. Disponível em: https://www.scopus.com/

[37] J. Prellberg e O. Kramer, «Lamarckian Evolution of Convolutional Neural Networks», em *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11102 LNCS, 2018, pp. 424–435. doi: 10.1007/978-3-319-99259-4_34.

[38] B. Fielding e L. Zhang, «Evolving Image Classification Architectures With Enhanced Particle Swarm Optimisation», *IEEE Access*, vol. 6, pp. 68560–68575, 2018, doi: 10.1109/ACCESS.2018.2880416.

[39] B. Fielding, T. Lawrence, e L. Zhang, «Evolving and Ensembling Deep CNN Architectures for Image Classification», em *2019 International Joint Conference on Neural Networks (IJCNN)*, IEEE, Jul. 2019, pp. 1–8. doi: 10.1109/IJCNN.2019.8852369.

[40] J. Huang, B. Xue, Y. Sun, e M. Zhang, «A Flexible Variable-length Particle Swarm Optimization Approach to Convolutional Neural Network Architecture Design», em *2021 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, Jun. 2021, pp. 934–941. doi: 10.1109/CEC45853.2021.9504716.

[41] J. Dong, L. Zhang, B. Hou, e L. Feng, «A Memetic Algorithm for Evolving Deep Convolutional Neural Network in Image Classification», em *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Dez. 2020, pp. 2663–2669. doi: 10.1109/SSCI47803.2020.9308162.

[42] Y. Sun, B. Xue, M. Zhang, e G. G. Yen, «Evolving Deep Convolutional Neural Networks for Image Classification», *IEEE Transactions on Evolutionary Computation*, vol. 24, n. 2, pp. 394–407, Abr. 2020, doi: 10.1109/TEVC.2019.2916183.

[43] D. A. Montecino, C. A. Perez, e K. W. Bowyer, «Two-Level Genetic Algorithm for Evolving Convolutional Neural Networks for Pattern Recognition», *IEEE Access*, vol. 9, pp. 126856–126872, 2021, doi: 10.1109/ACCESS.2021.3111175.

[44] B. Wang, B. Xue, e M. Zhang, «A transfer learning based evolutionary deep learning framework to evolve convolutional neural networks», em *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, New York, NY, USA: ACM, Jul. 2021, pp. 287–288. doi: 10.1145/3449726.3459455.

[45] Y. Xie, H. Chen, Y. Ma, e Y. Xu, «Automated design of CNN architecture based on efficient evolutionary search», *Neurocomputing*, vol. 491, pp. 160–171, Jun. 2022, doi: 10.1016/j.neucom.2022.03.046.

[46] T. Hassanzadeh, D. Essam, e R. Sarker, «EvoDCNN: An evolutionary deep convolutional neural network for image classification», *Neurocomputing*, vol. 488, pp. 271–283, Jun. 2022, doi: 10.1016/j.neucom.2022.02.003.

[47] G. Yuan, B. Wang, B. Xue, e M. Zhang, «Particle Swarm Optimization for Efficiently Evolving Deep Convolutional Neural Networks Using an Autoencoder-based Encoding Strategy», *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2023, doi: 10.1109/TEVC.2023.3245322.

[48] A. Jacotécole, J. Jacotécole, P. Fédérale De Lausanne, e F. Gabriel, «Neural Tangent Kernel: Convergence and Generalization in Neural Networks».

[49] M. J. Ali, L. Moalic, M. Essaid, e L. Idoumghar, «Designing Convolutional Neural Networks using Surrogate assisted Genetic Algorithm for Medical Image Classification», em *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, New York, NY, USA: ACM, Jul. 2023, pp. 263–266. doi: 10.1145/3583133.3590678.

[50] Z. Xun, L. Songbai, W. Ka-Chun, L. Qiuzhen, e T. Kaychen, «A Hybrid Search Method for Accelerating Convolutional Neural Architecture Search», em *Proceedings of the 2023 15th International Conference on Machine Learning and Computing*, em ICMLC '23. New York, NY, USA: ACM, Fev. 2023, pp. 177–182. doi: 10.1145/3587716.3587745.

[51] K. He, X. Zhang, S. Ren, e J. Sun, «Deep Residual Learning for Image Recognition», *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dez. 2015, doi: 10.1109/CVPR.2016.90.

[52] G. E. Hinton e R. R. Salakhutdinov, «Reducing the dimensionality of data with neural networks», *Science (1979)*, vol. 313, n. 5786, pp. 504–507, Jul. 2006, doi: 10.1126/SCIENCE.1127647/SUPPL_FILE/HINTON.SOM.PDF.

[53] A. Fischer e C. Igel, «An introduction to restricted Boltzmann machines», *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7441 LNCS, pp. 14–36, 2012, doi: 10.1007/978-3-642-33275-3_2/COVER.

[54] M. A. Keyvanrad e M. Mehdi Homayounpour, «A brief survey on deep belief networks and introducing a new object oriented toolbox (DeeBNet V3.0)», Acedido: 3 de Janeiro de 2024. [Em linha]. Disponível em: http://ceit.aut.ac.ir/~keyvanrad/DeeBNet%20Toolbox.html

[55] J. B. Pendry *et al.*, «References and Notes Supporting Online Material Reducing the Dimensionality of Data with Neural Networks», *IEEE Trans. Microw. Theory Tech*, vol. 47, n. 9, p. 653, 1999, doi: 10.1126/science.1129198.

[56] H. Lee, R. Grosse, R. Ranganath, e A. Y. Ng, «Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations», *Proceedings of the 26th International Conference On*

*Machine Learning, ICML 2009*, pp. 609–616, 2009, doi: 10.1145/1553374.1553453.

[57] H. Lee, C. Ekanadham, e A. Y. Ng, «Sparse deep belief net model for visual area V2».

[58] E. Real, A. Aggarwal, Y. Huang, e Q. V. Le, «Regularized Evolution for Image Classifier Architecture Search», *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, n. 01, pp. 4780–4789, Jul. 2019, doi: 10.1609/AAAI.V33I01.33014780.

[59] «CIFAR-10 and CIFAR-100 datasets». Acedido: 2 de Janeiro de 2024. [Em linha]. Disponível em: https://www.cs.toronto.edu/~kriz/cifar.html

[60] «Fashion MNIST». Acedido: 2 de Janeiro de 2024. [Em linha]. Disponível em: https://www.kaggle.com/datasets/zalando-research/fashionmnist

[61] «MNIST Dataset». Acedido: 2 de Janeiro de 2024. [Em linha]. Disponível em: https://www.kaggle.com/datasets/hojjatk/mnist-dataset

[62] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, e Y. Bengio, «An Empirical Evaluation of Deep Architectures on Problems with Many Factors of Variation».

[63] O. Russakovsky *et al.*, «ImageNet Large Scale Visual Recognition Challenge», Acedido: 2 de Janeiro de 2024. [Em linha]. Disponível em: http://image-net.org/challenges/LSVRC/

[64] «What is Python? Executive Summary | Python.org». Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: https://www.python.org/doc/essays/blurb/

[65] «TensorFlow». Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: https://www.tensorflow.org/?hl=pt-br

[66] «Keras: Deep Learning for humans». Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: https://keras.io/

[67] «Kaggle: Your Home for Data Science». Acedido: 31 de Dezembro de 2023. [Em linha]. Disponível em: https://www.kaggle.com/

[68] «Matplotlib — Visualization with Python». Acedido: 2 de Janeiro de 2024. [Em linha]. Disponível em: https://matplotlib.org/

[69] «ttiagojm/DBN-TF2: Deep Belief Networks in Tensorflow 2». Acedido: 25 de Janeiro de 2024. [Em linha]. Disponível em: https://github.com/ttiagojm/DBN-TF2

[70] F. Assunção, N. Lourenço, P. Machado, e B. Ribeiro, «Fast-DENSER++: Evolving Fully-Trained Deep Artificial Neural Networks», Mai. 2019, Acedido: 2 de Março de 2024. [Em linha]. Disponível em: https://arxiv.org/abs/1905.02969v1

[71] M. Suganuma, S. Shirakawa, e T. Nagao, «Designing Convolutional Neural Network Architectures Using Cartesian Genetic Programming», *Natural Computing Series*, pp. 185–208, 2020, doi: 10.1007/978-981-15-3685-4_7/COVER.

[72] K. D. Cooper e L. Torczon, «Parsers», *Engineering a Compiler*, pp. 83–159, Jan. 2012, doi: 10.1016/B978-0-12-088478-0.00003-7.

[73] J. F. Miller, «An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach», doi: 10.5555/2934046.2934074.

[74] J. F. Miller e S. L. Smith, «Redundancy and computational efficiency in cartesian genetic programming», *IEEE Transactions on Evolutionary Computation*, vol. 10, n. 2, pp. 167–174, Abr. 2006, doi: 10.1109/TEVC.2006.871253.

[75] «fillassuncao/fast-denser3: Fast Deep Evolutionary Network Structured Representation». Acedido: 3 de Março de 2024. [Em linha]. Disponível em: https://github.com/fillassuncao/fast-denser3

[76] «sg-nm/cgp-cnn-PyTorch: A Genetic Programming Approach to Designing CNN Architectures, In GECCO 2017 (oral presentation, Best Paper Award)». Acedido: 10 de Maio de 2024. [Em linha]. Disponível em: https://github.com/sg-nm/cgp-cnn-PyTorch

[77] S. Ioffe e C. Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, Fev. 2015, Acedido: 12 de Maio de 2024. [Em linha]. Disponível em: https://arxiv.org/abs/1502.03167v3

[78] K. He, X. Zhang, S. Ren, e J. Sun, «Deep Residual Learning for Image Recognition», Acedido: 3 de Março de 2024. [Em linha]. Disponível em: http://image-net.org/challenges/LSVRC/2015/

[79] «CGP C0nvSet NASWOT - CIFAR10». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EcdmpNvGfN5Fl5_dLbVjEM8BA6iCEBT9rDDBMucgU9JIfw?e=yPNRcZ

[80] «CGP ConvSet NTK - CIFAR10». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EeEHeXJEia1Ci3ZUNwlGOpwB2xQXwbCIRGYWMhjycaQXGA?e=dj7xxu

[81] «CGP ResSet NASWOT - CIFAR10». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-

my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EVj-cfd0FpZNjbwAr6Kj8QcB8Pql1_Ht2V0yD9433ff_wg?e=N2HciS

[82] «CGP ResSet NTK - CIFAR10». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EbV59V6UyuhEjxeM3A7y0SIBpbIuX4bSrAPfObfyBfOxOQ?e=7Z0bBj

[83] M. Lin, Q. Chen, e S. Yan, «Network In Network», *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*, Dez. 2013, Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://arxiv.org/abs/1312.4400v3

[84] «Fast-Denser++ NASWOT - CIFAR100». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/ET8p0z_00XhEveUN43AbCpoB0r0Jq8Jbti8Bdq7axxF8ZQ?e=XPc2N9

[85] «CGP NASWOT - CIFAR100». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EV-65yi_wBBCh2D1NKWn17cBXjwKVQ6GOY_C1BIeYHOG5A?e=xNV98r

[86] «CGP NASWOT Search Global Avg Pooling - CIFAR100». Acedido: 4 de Junho de 2024. [Em linha]. Disponível em: https://ipcbcampus-my.sharepoint.com/:i:/g/personal/tiagomartins1_ipcbcampus_pt/EUbJBmWy_bdAjUfn1cOgj5MB3y8UGdrx5raI39iB-3RZRg?e=5mBuv0