Optimizing Complex Neural Networks with Population-Based Genetic Algorithms

Mateus de Freitas Rosa[1], **Murillo Guimarães Carneiro**[1]

[1] Faculdade de Computação, Universidade Federal de Uberlândia (UFU)
Av. João Naves de Ávila, nº 2121, Bairro Santa Mônica
CEP 38.400-902 – Uberlândia – MG – Brasil

{mateusfreitasrosa, murillo.carneiro}@ufu.br

Abstract. *The design of Convolutional Neural Network (CNN) architectures is a complex and costly task, requiring considerable expertise and experimentation. Neural Architecture Search (NAS) aims to automate this process. This work presents a NAS system based on Genetic Algorithms (GA) that employs a graph-based representation for network architectures. The system uses customized genetic operators to directly manipulate the network topology, including convolutional and dense layers. The fitness function considers both performance metrics and the number of parameters, aiming to balance effectiveness and efficiency. To reduce training time for unpromising architectures, an early stopping mechanism based on batches was developed. Mechanisms such as elitism and automatic correction of invalid graphs (e.g., cycles, incompatible dimensions) are also integrated into the evolutionary process. Experiments conducted on the CIFAR-10 and SVHN datasets demonstrate the system's ability to evolve competitive architectures. The main contribution is a robust and adaptable framework for exploring the neural architecture search space.*

Resumo. *O projeto de arquiteturas de Redes Neurais Convolucionais (CNNs) é uma tarefa complexa e dispendiosa, exigindo considerável experiência e experimentação. O Neural Architecture Search (NAS) visa automatizar este processo. Este trabalho apresenta um sistema de NAS baseado em Algoritmos Genéticos (AG) que utiliza uma representação em grafo para as arquiteturas. O sistema emprega operadores genéticos customizados para a manipulação direta da topologia da rede, abrangendo camadas convolucionais e densas. A função de aptidão considera tanto a métrica de desempenho quanto o número de parâmetros, buscando um equilíbrio entre eficácia e eficiência. Para reduzir o tempo de treinamento de arquiteturas pouco promissoras, foi desenvolvido um mecanismo de early stopping baseado em lotes. Mecanismos como elitismo e correção automática de invalidez de grafos (e.g., ciclos, dimensões incompatíveis) também são integrados ao processo evolutivo. Experimentos conduzidos nos conjuntos de dados CIFAR-10 e SVHN demonstram a capacidade do sistema em evoluir arquiteturas competitivas. A principal contribuição é um framework robusto e adaptável para a exploração do espaço de arquiteturas neurais.*

1. Introdução

A Inteligência Artificial (IA) tem alcançado avanços expressivos nas últimas décadas, especialmente em áreas como Visão Computacional [He et al. 2016], Processamento de Linguagem Natural [Devlin et al. 2019] e Sistemas de Recomendação [Covington et al. 2016], impulsionada principalmente pelo uso de Modelos Neurais Profundos. Redes Neurais Convolucionais (CNNs), Redes Recorrentes (RNNs) e Transformers têm se destacado por sua capacidade de capturar padrões complexos em dados de diferentes naturezas, demonstrando alto desempenho em tarefas de classificação, detecção, segmentação e geração de texto.

Entretanto, o desenvolvimento de uma arquitetura neural eficaz para um problema específico permanece uma tarefa desafiadora. Esse processo normalmente exige experiência prévia, conhecimento empírico e sucessivos ciclos de tentativa e erro. Na prática, define-se uma arquitetura inicial — composta por camadas convolucionais capazes de extrair representações relevantes e camadas densas que aprendem padrões a partir dessas representações — seguida de testes por múltiplas épocas de treinamento. Mesmo quando um modelo atinge bons resultados, é comum que a busca por alternativas promissoras seja interrompida precocemente, o que pode impedir a descoberta de soluções ainda mais eficazes [Elsken et al. 2019].

Esse cenário evidencia a necessidade de métodos automáticos para a descoberta de arquiteturas neurais, surgindo assim o campo de Neural Architecture Search (NAS). O NAS busca arquiteturas otimizadas por meio de algoritmos baseados em técnicas como aprendizado por reforço [Zoph and Le 2016], diferenciação contínua [Liu et al. 2018] e algoritmos evolucionários [Real et al. 2019]. Estas abordagens buscam automatizar o processo de experimentação, reduzindo o custo computacional e o tempo investido em buscas manuais.

Diferentemente de abordagens que utilizam técnicas de otimização baseadas em aprendizado por reforço [Zoph and Le 2016] ou diferenciação contínua [Liu et al. 2018], este trabalho propõe uma abordagem de Neural Architecture Search (NAS) baseada em população, utilizando algoritmos genéticos como otimizador no processo de busca por arquiteturas mais eficazes. O método explora operadores evolutivos clássicos — seleção, mutação e crossover — aplicados sobre uma população de arquiteturas candidatas, permitindo a evolução iterativa de soluções ao longo de múltiplas gerações.

Como diferencial, as arquiteturas são representadas como grafos direcionados acíclicos, o que oferece maior flexibilidade na manipulação estrutural e possibilita a construção de arquiteturas não lineares, como redes residuais com conexões de atalho (skip connections) [He et al. 2016]. Além disso, o sistema implementa um espaço de busca dinâmico, em que o conjunto de operadores, profundidade e largura da rede podem ser definidos no momento da execução, adaptando-se ao problema-alvo.

A proposta suporta a geração automática de arquiteturas de Redes Neurais Convolucionais (CNNs), utilizando as camadas Conv2D (operações de convolução bidimensional para extração de padrões visuais) e Linear (camadas totalmente conectadas para mapeamento de características em saídas) da biblioteca PyTorch [Paszke 2019]. Essa modularidade, aliada à flexibilidade topológica, torna a abordagem escalável e compatível com tarefas complexas em Visão Computacional.

Para avaliar a eficácia da abordagem proposta, serão utilizados dois conjuntos de dados com diferentes níveis de complexidade: CIFAR-10 (nível intermediário) e SVHN (nível avançado). Cada um desses datasets apresenta desafios específicos, permitindo testar a capacidade do algoritmo de NAS baseado em população em adaptar-se a diferentes cenários e encontrar arquiteturas eficazes para cada caso.

2. Referencial Teórico

O processo de *Neural Architecture Search* (NAS) tem sido amplamente explorado na literatura, com diversas abordagens propostas para automatizar a descoberta de arquiteturas neurais eficazes. Dentre essas abordagens, destacam-se três principais vertentes: diferenciação contínua, aprendizado por reforço e algoritmos evolutivos.

A diferenciação contínua, como proposta em DARTS [Liu et al. 2018], permite a otimização direta das estruturas da rede neural por meio do uso de gradiente, viabilizando uma busca eficiente em um espaço relaxado de arquiteturas. Essa técnica reduziu significativamente o custo computacional associado ao processo de busca.

Por outro lado, o aprendizado por reforço, introduzido por Zoph e Le [Zoph and Le 2016], modela o NAS como um problema de tomada de decisão sequencial, onde um controlador (geralmente uma RNN) gera arquiteturas candidatas que são avaliadas com base em recompensas derivadas do desempenho dos modelos gerados.

Já os algoritmos evolutivos, como apresentado por Real et al. [Real et al. 2019], utilizam operadores inspirados na evolução natural — como mutação, crossover e seleção — para explorar populações de arquiteturas e promover diversidade na busca por soluções mais promissoras.

Entre essas abordagens, destaca-se ainda o *RelativeNAS*, proposto por Tan et al. [Tan et al. 2021], que combina elementos das técnicas de diferenciação contínua com princípios evolutivos. Essa abordagem híbrida baseia-se em um mecanismo de aprendizado lento-rápido (*slow-fast learning*) e avaliação relativa de desempenho entre arquiteturas, permitindo uma exploração mais eficiente do espaço de busca sem depender exclusivamente de avaliações absolutas e exaustivas.

Esses trabalhos representam os principais pilares do NAS na literatura e fundamentam a proposta deste trabalho, que adota uma abordagem evolucionária com representações flexíveis em grafos acíclicos direcionados e um espaço de busca adaptável ao problema.

Como diferencial, também introduzimos um processo de reconstrução parcial da arquitetura convolucional, aplicado somente aos trechos afetados por operações de mutação ou crossover. Em vez de reconstruir toda a rede ou inserir adaptadores intermediários — o que frequentemente leva à criação de redes excessivamente profundas e ineficientes — a proposta consiste em transformar as camadas conflitantes em camadas do tipo *lazy* (camadas "preguiçosas", como `LazyConv2D` ou `LazyLinear`). Essas camadas adiam a definição explícita de suas dimensões de entrada até o momento da execução, resolvendo automaticamente incompatibilidades estruturais.

Além disso, a representação da arquitetura como um grafo direcionado acíclico permite a construção de topologias não sequenciais, incluindo o uso de conexões de atalho (*skip connections*) [He et al. 2016], o que contribui para o fluxo de gradientes e a

estabilidade do treinamento.

3. Metodologia

Nesta seção, é apresentada a metodologia utilizada para a implementação da abordagem proposta. Toda a implementação foi realizada em Python 3.10, utilizando bibliotecas atualizadas e voltadas ao desenvolvimento de redes neurais e algoritmos evolucionários. Para a definição e treinamento das arquiteturas convolucionais, foi empregada a biblioteca PyTorch [Paszke 2019], que oferece flexibilidade na construção modular de camadas, favorecendo a manipulação estrutural das redes.

3.1. Representação Genética das Arquiteturas

Em algoritmos genéticos, a representação genética dos indivíduos é um aspecto fundamental para o sucesso dos operadores evolutivos, como mutação, crossover e seleção. Neste trabalho, cada indivíduo da população corresponde a uma arquitetura de rede neural convolucional, cuja estrutura é representada por um grafo direcionado acíclico (DAG — *Directed Acyclic Graph*).

Cada nó do grafo representa uma camada funcional da rede, podendo ser uma camada convolucional (`Conv2D`) ou totalmente conectada (`Linear`). Para evitar conflitos de dimensionamento entre camadas conectadas — problema comum em arquiteturas geradas dinamicamente — todas as camadas são inicialmente instanciadas como versões "preguiçosas"da biblioteca PyTorch, como `LazyConv2D` e `LazyLinear`. Essas camadas adiam a definição de suas dimensões de entrada até o momento em que os dados reais são propagados pela rede, permitindo maior flexibilidade na recomposição da arquitetura durante os processos evolutivos.

Essa abordagem reduz significativamente a complexidade algorítmica relacionada ao encaixe de tensores entre camadas consecutivas e facilita a construção de arquiteturas não lineares. Após sua criação, cada camada (nó) permanece inalterada durante o tempo de vida do indivíduo, garantindo estabilidade na propagação das informações e consistência nas operações de mutação parcial.

3.2. Espaço de Busca

O espaço de busca definido neste trabalho concentra-se nos parâmetros essenciais das camadas convolucionais (`Conv2D`) e densas (`Linear`), de forma a garantir um equilíbrio entre diversidade arquitetural e viabilidade computacional. Essa configuração permite que o algoritmo explore uma variedade significativa de combinações estruturais, respeitando limites práticos de complexidade e desempenho.

Para as camadas `Conv2D`, os seguintes hiperparâmetros são considerados: número de filtros (`out_channels`), tamanho do kernel (`kernel_size`), passo (`stride`), preenchimento (`padding`) e a função de ativação. Já nas camadas densas, são definidos o número de neurônios de saída (`out_features`), a função de ativação e a taxa de *dropout*. A Tabela 1 apresenta os valores discretos considerados para cada parâmetro no espaço de busca utilizado.

Tabela 1. Parâmetros considerados no espaço de busca.

Parâmetro	Valores possíveis
`Conv2D.out_channels`	[16, 32, 64, 128, 256]
`Conv2D.kernel_size`	[3, 5, 7]
`Conv2D.stride`	[1, 2]
`Conv2D.padding`	[0, 1, 2]
`Conv2D.activation`	[”ReLU”, ”LeakyReLU”, ”GELU”, ”SiLU”]
`Linear.out_features`	[64, 128, 256, 512, 1024]
`Linear.activation`	[”ReLU”, ”LeakyReLU”, ”GELU”, ”SiLU”]
`Linear.dropout_p`	[0.0, 0.1, 0.2, 0.3, 0.4]

A implementação permite que esses intervalos de valores sejam definidos diretamente durante a instanciação do objeto responsável pela busca evolutiva, o que torna o sistema flexível e adaptável a diferentes domínios de aplicação. Essa funcionalidade também possibilita a incorporação de conhecimento prévio ou empírico sobre a tarefa-alvo, orientando o processo evolutivo para regiões promissoras do espaço de busca.

3.3. Operadores Evolutivos

Neste trabalho, são implementados os principais operadores genéticos utilizados em algoritmos evolucionários: seleção, crossover, mutação e elitismo. Esses operadores atuam em conjunto para promover a evolução das arquiteturas ao longo das gerações, equilibrando exploração e preservação de boas soluções.

A operação de *crossover* tem como objetivo recombinar trechos de arquiteturas de dois indivíduos (pais), promovendo a geração de novas combinações potencialmente promissoras. Já a *mutação* introduz modificações pontuais nas arquiteturas, permitindo variações estruturais que favorecem a descoberta de soluções inovadoras.

A aplicação desses operadores é guiada por um método de seleção, que pode ser baseado em roleta, torneio ou outras estratégias estocásticas. Inicialmente, indivíduos são selecionados conforme o critério adotado. Em seguida, pares desses indivíduos passam pelo operador de crossover, resultando na geração de dois novos indivíduos (filhos). Após o crossover, aplica-se o operador de mutação sobre os filhos gerados, introduzindo variações controladas nas novas arquiteturas.

Para garantir a preservação das melhores soluções ao longo do processo evolutivo, também é adotado o operador de *elitismo*, responsável por manter inalterados os melhores indivíduos da geração atual. Isso assegura que boas soluções não sejam perdidas, mesmo que a nova geração apresente desempenho inferior.

3.3.1. Crossover

O operador de *crossover* (cruzamento) implementado neste trabalho atua sobre dois indivíduos selecionados da população. Inicialmente, define-se aleatoriamente qual parte da arquitetura será alvo da recombinação: o bloco convolucional ou o bloco denso (composto por camadas lineares).

Para a região selecionada, um ponto de corte é escolhido aleatoriamente em cada um dos pais. A partir desses pontos, são gerados dois novos indivíduos: o primeiro filho herda a porção inicial do primeiro pai e a porção final do segundo; o segundo filho realiza a operação inversa, combinando a parte inicial do segundo pai com a parte final do primeiro. Esse processo é ilustrado na Figura 1.

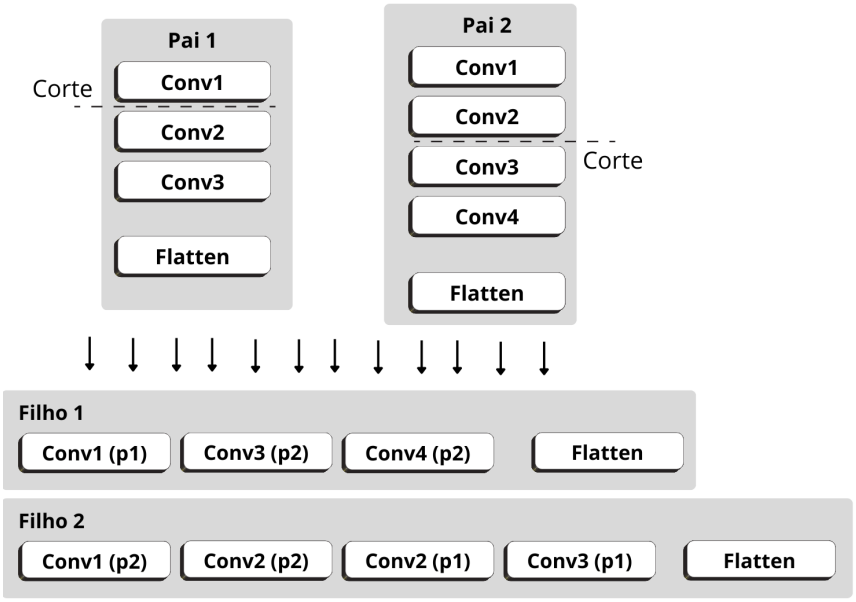

Figura 1. Exemplo do operador de *crossover* aplicado à parte convolucional de dois indivíduos. Dois filhos são gerados a partir da recombinação de blocos convolucionais com base em um ponto de corte aleatório.

Após a recombinação da parte-alvo (convolucional ou densa), o restante da arquitetura é herdado diretamente de cada respectivo pai: o filho 1 recebe a parte complementar do pai 1, enquanto o filho 2 herda a parte complementar do pai 2. Por exemplo, caso a parte recombinada seja a convolucional, o bloco de camadas densas será herdado integralmente de cada pai correspondente. O mesmo se aplica de forma análoga quando o crossover ocorre na parte densa.

Contudo, a combinação de partes distintas de arquiteturas pode gerar conflitos estruturais, especialmente relacionados à incompatibilidade de dimensões entre as camadas. Isso ocorre quando a saída da última camada da primeira parte (vinda de um dos pais) não é compatível com a entrada da primeira camada da segunda parte (vinda do outro pai). Para resolver esse problema, as primeiras camadas da parte herdada são automaticamente reinicializadas como camadas `LazyConv2D` ou `LazyLinear`, dependendo do tipo de bloco. Essas camadas *lazy* adiam a definição da forma de entrada até a primeira passagem dos dados, o que garante a compatibilidade dimensional sem necessidade de ajustes manuais.

Além disso, quando o crossover afeta o bloco convolucional, a alteração impacta diretamente a saída da camada `Flatten`, que serve como entrada para a primeira camada linear. Nesse cenário, é obrigatório também reinicializar a primeira camada linear como `LazyLinear`, pois a alteração no volume de saída do bloco convolucional modifica a dimensão esperada pela camada densa subsequente.

A Figura 2 ilustra esse mecanismo de reconstrução pontual em dois cenários distintos: recombinação da parte convolucional (acima) e da parte densa (abaixo). As camadas que precisam ser reconstruídas estão destacadas em vermelho, evidenciando os pontos de adaptação estrutural automática.

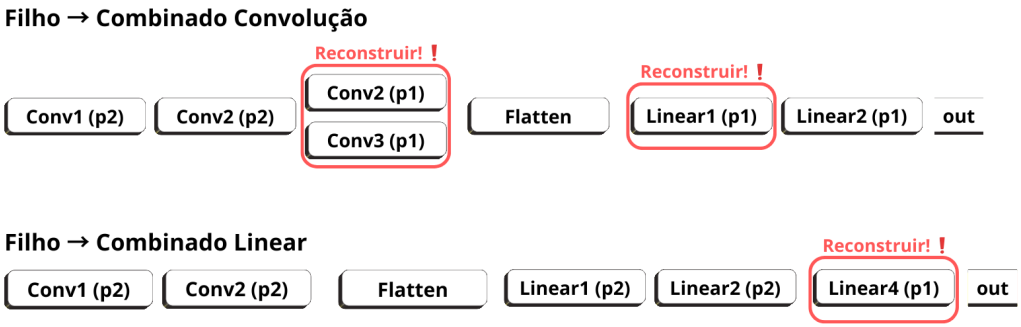

Figura 2. Exemplos de reconstrução estrutural após crossover. Acima: recombinação do bloco convolucional exige reconstrução das camadas convolucionais herdadas e da primeira camada densa. Abaixo: recombinação do bloco denso exige reinicialização da primeira camada herdada da parte final.

Essa abordagem torna o processo de recombinação mais robusto, permitindo a geração de arquiteturas viáveis mesmo diante de estruturas heterogêneas, ao mesmo tempo em que preserva a integridade das conexões entre os blocos convolucional e denso.

Essa abordagem torna o processo de recombinação mais robusto, permitindo a geração de arquiteturas viáveis mesmo diante de estruturas heterogêneas, ao mesmo tempo em que preserva a integridade das conexões entre os blocos convolucional e denso.

No entanto, mesmo com esse mecanismo de reconstrução automática por meio de camadas *lazy*, observa-se que, com o aumento da complexidade das arquiteturas ao longo das gerações, alguns conflitos estruturais ainda podem surgir. Isso se deve à ampliação do número de camadas e à diversidade de combinações entre estruturas cada vez mais distintas.

Para mitigar esses efeitos e garantir a consistência dos indivíduos gerados, o algoritmo realiza, ao final de cada geração, uma etapa adicional de validação e correção estrutural. Nessa etapa, os indivíduos recém-gerados são analisados quanto à compatibilidade entre suas camadas, e quaisquer inconsistências remanescentes são corrigidas automaticamente. Esse processo assegura que todos os indivíduos da nova geração estejam aptos a serem avaliados e treinados, mantendo a integridade do processo evolutivo.

3.3.2. Mutação

O operador de *mutação* tem como objetivo introduzir variações pontuais nas arquiteturas dos indivíduos, promovendo a diversidade genética da população e favorecendo a exploração de novas regiões no espaço de busca. Neste trabalho, foram implementadas quatro operações básicas de mutação:

- Adição de uma nova camada;

- Remoção de uma camada existente;
- Adição de uma conexão de atalho (*skip connection*);
- Remoção de uma conexão de atalho.

As operações de adição são, em geral, mais diretas, consistindo na inserção de uma nova camada convolucional ou linear em uma posição aleatória da arquitetura, ou na criação de uma nova conexão entre dois nós não consecutivos do grafo, respeitando a aciclicidade da estrutura.

Por outro lado, as operações de remoção exigem maior cautela, uma vez que a exclusão de camadas ou conexões pode comprometer a integridade da arquitetura. Para garantir que a estrutura resultante permaneça funcional, é realizada uma análise prévia dos possíveis candidatos à remoção. Apenas camadas ou conexões cuja exclusão não interrompa o fluxo de dados — ou seja, que não quebrem o caminho do grafo entre a entrada e a saída — são consideradas elegíveis para remoção.

Essa verificação estrutural assegura que o operador de mutação não gere arquiteturas inválidas, preservando a viabilidade dos indivíduos ao longo do processo evolutivo.

3.4. Avaliação e Fitness

A avaliação de cada indivíduo da população é realizada com base em uma métrica de desempenho definida previamente, como acurácia, *F1-score*, *precision* ou *recall*, dependendo da natureza do problema e da base de dados utilizada. Essa métrica é utilizada como principal indicador da qualidade da arquitetura neural.

Além disso, a avaliação incorpora um fator de penalização proporcional à quantidade de parâmetros da arquitetura. Essa estratégia visa favorecer arquiteturas mais compactas e eficientes, promovendo um equilíbrio entre desempenho e complexidade computacional.

O impacto dessa penalização é controlado por um coeficiente configurável, que define o peso relativo da quantidade de parâmetros em relação à métrica principal. Esse fator deve ser cuidadosamente ajustado: quando definido como muito alto, pode induzir o modelo a priorizar arquiteturas excessivamente simples em detrimento da performance; quando muito baixo, torna-se ineficaz na regulação do tamanho das redes. Por essa razão, recomenda-se que a penalização seja sutil, atuando apenas como um critério de desempate entre arquiteturas com desempenhos similares ou muito próximo.

Essa função de *fitness* balanceada permite guiar o processo evolutivo de forma eficiente, encorajando a descoberta de arquiteturas com bom desempenho preditivo e, ao mesmo tempo, menor custo computacional.

4. Experimentos e Resultados

4.1. Configuração Experimental

Para a validação da abordagem proposta, foram realizados experimentos em dois conjuntos de dados amplamente utilizados em tarefas de classificação de imagens: CIFAR-10 e SVHN.

O **CIFAR-10** é composto por 60.000 imagens coloridas ($3 \times 32 \times 32$ pixels), divididas em 10 categorias: avião, automóvel, pássaro, gato, veado, cachorro, sapo, cavalo,

navio e caminhão. Para reduzir a complexidade computacional e padronizar a entrada das arquiteturas, as imagens foram convertidas para escala de cinza, resultando em dimensões $1 \times 32 \times 32$.

O **SVHN** (Street View House Numbers) contém mais de 600.000 imagens coloridas de números encontrados em placas de residências, coletadas a partir do Google Street View. As imagens também possuem resolução de 32×32 pixels e 3 canais. Assim como no CIFAR-10, foi aplicado um pré-processamento para convertê-las para tons de cinza, com dimensões finais de $1 \times 32 \times 32$.

Hiperparâmetros do algoritmo genético: A Tabela 2 resume os principais hiperparâmetros utilizados no algoritmo genético:

Tabela 2. Configuração dos hiperparâmetros utilizados no algoritmo genético.

Parâmetro	**Valor**
Tamanho da população	100
Número máximo de gerações	100
Taxa de mutação	0,15
Número de pais para cruzamento (*crossover*)	30
Número de elites preservados	10
Método de seleção	Torneio (*tournament*)
Taxa de reinserção	0,07
Máximo de camadas convolucionais	6
Máximo de camadas densas	10
Tolerância para parada antecipada (*early stopping*)	360 *batches* sem melhora
Critério de parada	Parada antecipada por *batch*
Taxa de aprendizado (*learning rate*)	0,001
Limite de parâmetros por arquitetura	3 milhões
Peso da acurácia na métrica de avaliação (*fitness*)	1
Peso da penalização por número de parâmetros	0,005

4.2. Espaço de busca:

Tabela 3. Espaço de busca utilizado para camadas convolucionais e densas no presente trabalho.

Parâmetro	**Valores possíveis**
`Conv2D.out_channels`	[16, 32, 64]
`Conv2D.kernel_size`	[1, 2, 3, 4, 5]
`Conv2D.stride`	[1, 2, 3, 4]
`Conv2D.padding`	[0, 1, 2, `'valid'`]
`Conv2D.dilation`	[1, 2, 3, 4]
`Conv2D.activation`	`ReLU`, `LeakyReLU`, `GELU`, `Swish`, `Tanh`, `Sigmoid`
`Linear.out_features`	[32, 64, 128, 256]
`Linear.activation`	mesmas funções das convolucionais
`Linear.dropout_p`	[nenhum, 0.1, 0.3, 0.5]

4.3. Configuração da arquitetura:

Foi estabelecido um limite máximo de 6 camadas convolucionais e 10 camadas densas iniciais por arquitetura. O número total de parâmetros é restringido a, no máximo, 2 milhões. Durante a construção das redes, o espaço de busca considerado inclui uma variedade de tamanhos de filtros, funções de ativação e configurações de dropout, conforme detalhado na Tabela 1.

4.4. Treinamento e avaliação:

Durante a avaliação de cada indivíduo, o modelo é treinado por até 5 épocas com um *batch size* de 64. A taxa de aprendizado padrão utilizada é 0.01, sendo reduzida para 0.001 durante o ajuste fino. O treinamento é interrompido precocemente caso não haja melhora após duas iterações consecutivas, com tolerância de 0.01. Também foi implementado *early stopping* por batch, com paciência de 120 lotes e tolerância de $1e^{-5}$.

A função de fitness combina a métrica de avaliação (acurácia, neste experimento) com um fator de penalização proporcional ao número de parâmetros da arquitetura. Os pesos utilizados foram 1.0 para a métrica e 0.001 para a penalização, conforme explicado na Seção 3.4.

Infraestrutura computacional: Todos os experimentos foram conduzidos utilizando a biblioteca PyTorch, com suporte a aceleração por GPU via CUDA. A configuração do dispositivo é automaticamente ajustada entre `cuda` e `cpu`, conforme a disponibilidade do hardware. As execuções ocorreram em uma estação com sistema operacional **Windows**, equipada com uma **GPU NVIDIA GeForce RTX 4060 Ti (8 GB de VRAM)** e 16 GB de memória RAM, sendo suficiente para suportar o treinamento das arquiteturas ao longo de todas as gerações do algoritmo.

4.5. Resultados

Nesta seção, são apresentados os principais resultados obtidos com o algoritmo proposto ao longo dos experimentos conduzidos nas bases de dados MNIST, Fashion-MNIST e CIFAR-10.

Tabela 4. Desempenho da arquitetura proposta em comparação com modelos de referência no CIFAR-10.

Modelo	Acurácia (%)	Parâmetros	Fonte
Arquitetura Proposta	97.61	**22.298**	Este trabalho
NAS v3 + mais filtros	96.35	37.4M	[Zoph and Le 2016]
DenseNet-BC (L=100, k=24)	94.81	15.3M	[Huang et al. 2017]
DenseNet-BC (L=100, k=12)	94.08	0.8M	[Huang et al. 2017]
NAP (Ding et al., 2022)	97.52	3.07M	[Ding et al. 2022]
CDARTS (Yu et al., 2022)	98.32	—	[Yu et al. 2022]
AmoebaNet-B (18, 512)	**99.00**	556M	[Huang et al. 2019]

Tabela 5. Desempenho da arquitetura proposta no conjunto SVHN comparado com modelos baseados em CNN.

Modelo	Acurácia (%)	Parâmetros	Fonte
Arquitetura Proposta	91.48	**389,610**	Este trabalho
WRN-16-4 (dp)	**98.36**	∼2.7M	[Zagoruyko and Komodakis 2016]
ZARTS	97.55	—	[Wang et al. 2022]

Vale destacar que o tempo de treinamento por arquitetura foi significativamente reduzido devido à aplicação de um mecanismo de *early stopping* por **lote (batch)**, o qual interrompe o treinamento de modelos não promissores de forma antecipada. Esse processo resultou em uma economia substancial de tempo, totalizando cerca de **7,4 horas** para a avaliação de **100 gerações com 100 indivíduos cada**.

5. Considerações finais e direções futuras

Os resultados obtidos indicam que o algoritmo proposto é capaz de gerar arquiteturas altamente eficazes com um número reduzido de parâmetros, superando ou se aproximando de modelos clássicos reconhecidos na literatura. A eficiência no tempo de treinamento e a simplicidade da abordagem evidenciam o potencial da busca evolutiva mesmo em espaços de busca amplos e sem restrições rígidas.

O processo de busca por melhores arquiteturas nos conjuntos CIFAR-10 e SVHN foi realizado em aproximadamente 7 a 12 horas, utilizando um computador com GPU NVIDIA GeForce RTX 4060 Ti, 16 GB de memória RAM e processador Intel Core i5 de 13ª geração. Considerando essas especificações, é plausível supor que, com maior poder computacional ou tempo de processamento estendido, o algoritmo NAS poderia encontrar arquiteturas ainda mais eficazes, potencialmente superando os resultados aqui apresentados.

Como trabalho futuro, pretende-se aplicar o algoritmo proposto em tarefas mais complexas do que as abordadas neste trabalho, incluindo desafios como segmentação de imagens e reconhecimento em séries temporais, em que o uso de redes convolucionais tem se mostrado eficaz.

Agradecimentos

Os autores agradecem o apoio financeiro dado pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (projetos n. 420212/2023-0 e 445027/2024-0).

Referências

Covington, P., Adams, J., and Sargin, E. (2016). Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186.

Ding, Y., Wu, Y., Huang, C., Tang, S., Wu, F., Yang, Y., Zhu, W., and Zhuang, Y. (2022). Nap: Neural architecture search with pruning. *Neurocomputing*, 477:85–95.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32.

Liu, H., Simonyan, K., and Yang, Y. (2018). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Paszke, A. (2019). Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2019). Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789.

Tan, H., Cheng, R., Huang, S., He, C., Qiu, C., Yang, F., and Luo, P. (2021). Relative-nas: Relative neural architecture search via slow-fast learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(1):475–489.

Wang, X., Guo, W., Su, J., Yang, X., and Yan, J. (2022). Zarts: On zero-order optimization for neural architecture search. *Advances in Neural Information Processing Systems*, 35:12868–12880.

Yu, H., Peng, H., Huang, Y., Fu, J., Du, H., Wang, L., and Ling, H. (2022). Cyclic differentiable architecture search. *IEEE transactions on pattern analysis and machine intelligence*, 45(1):211–228.

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. *arXiv preprint arXiv:1605.07146*.

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.