RELATÓRIO TÉCNICO

Tech Challenge - Fase 2

Deep Learning com Algoritmo Genético para Classificação de Dígitos Manuscritos

 -IADT

### Autores:

Diego Silva Prado - RM364919

Francisco Ferreira de Araújo - RM361133 Ricardo Almeida da Rocha - RM362655

**Professor(a):** Poliana Nascimento Ferreira

Link para o github: <https://github.com/araujofran/digits-ag-dl>

Link para o vídeo: <https://youtu.be/Ac_xoc6AQzQ>

Data de Entrega: 05 de Julho de 2025

[Introdução 3](#_Toc205246477)

[Exploração de Dados ( do DataSet Digits) 4](#_Toc205246478)

[Visão Geral do Fluxo Computacional 11](#_Toc205246479)

[Pré-processamento (do dataset Digits) 14](#_Toc205246480)

[Modelagem (da Rede Neural) 21](#_Toc205246481)

[Treinamento e Avaliação (do Modelo Híbrido AG+DL) 33](#_Toc205246482)

[Avaliação de Baseline 47](#_Toc205246483)

[Análise dos Resultados: 50](#_Toc205246484)

[Insights Obtidos 50](#_Toc205246485)

[Conclusão Final 51](#_Toc205246486)

# Introdução

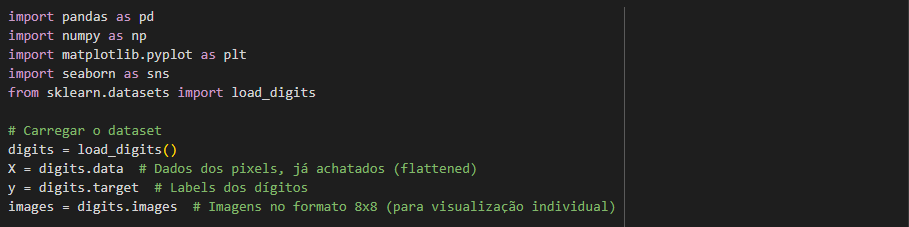
Este relatório apresenta o desenvolvimento e implementação de um modelo preditivo híbrido que integra Deep Learning (DL) e Algoritmos Genéticos (AG). O projeto, desenvolvido como parte do Tech Challenge da FIAP-IADT, tem como objetivo otimizar os pesos de uma camada específica de uma Rede Neural Multicamadas (MLP) para classificação de dígitos manuscritos, utilizando o dataset digits do Scikit-learn.

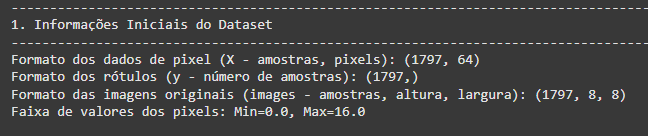
A metodologia implementada compreende: (1) preparação e pré-processamento dos dados, (2) definição da arquitetura da Rede Neural, (3) otimização híbrida através de AG para uma camada específica combinada com retropropagação (RMSprop) para as demais camadas, e (4) avaliação quantitativa dos resultados. O objetivo principal é investigar a eficácia do Algoritmo Genético na otimização de pesos neurais, enquanto as camadas subsequentes são ajustadas por métodos de otimização baseados em gradiente.

O documento está estruturado nas seguintes seções: exploração e pré-processamento dos dados, modelagem da arquitetura neural, implementação do algoritmo genético, treinamento e avaliação do modelo híbrido, e análise dos resultados obtidos.

# Exploração de Dados ( do DataSet Digits)

Esta seção apresenta a análise exploratória do dataset digits, com o objetivo de caracterizar sua estrutura, variáveis e propriedades estatísticas. O processo iniciou-se com a importação das bibliotecas necessárias (pandas, numpy, matplotlib, seaborn, sklearn) e o carregamento do dataset digits da biblioteca Scikit-learn. Este dataset contém imagens de dígitos manuscritos (0 a 9) representadas como vetores de 64 pixels (imagens 8x8) com seus respectivos rótulos de classe.



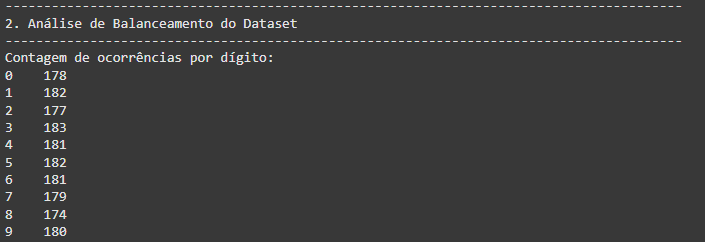


Ao carregar os dados, observamos as seguintes características:

* Formato dos Dados: Matriz de 1797 amostras × 64 características, onde cada valor representa a intensidade de um pixel (escala 0-16);
* Rótulos (Targets): Vetor de 1797 elementos contendo as classes dos dígitos (0-9);
* Imagens: Imagens 8×8 pixels em escala de cinza.

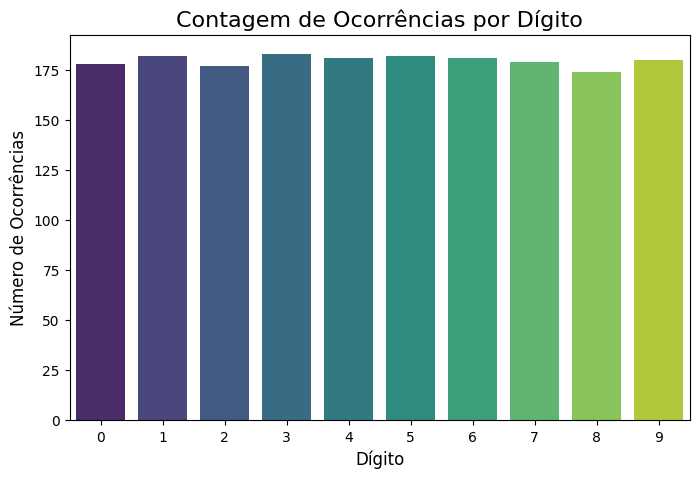
**Análise do Balanceamento do Dataset**

A distribuição das classes foi analisada para verificar possível desbalanceamento. Os resultados demonstram uma distribuição aproximadamente uniforme entre os dígitos, com variações mínimas nas frequências das classes. Esta característica é adequada para algoritmos de classificação, minimizando viés de classe durante o treinamento.



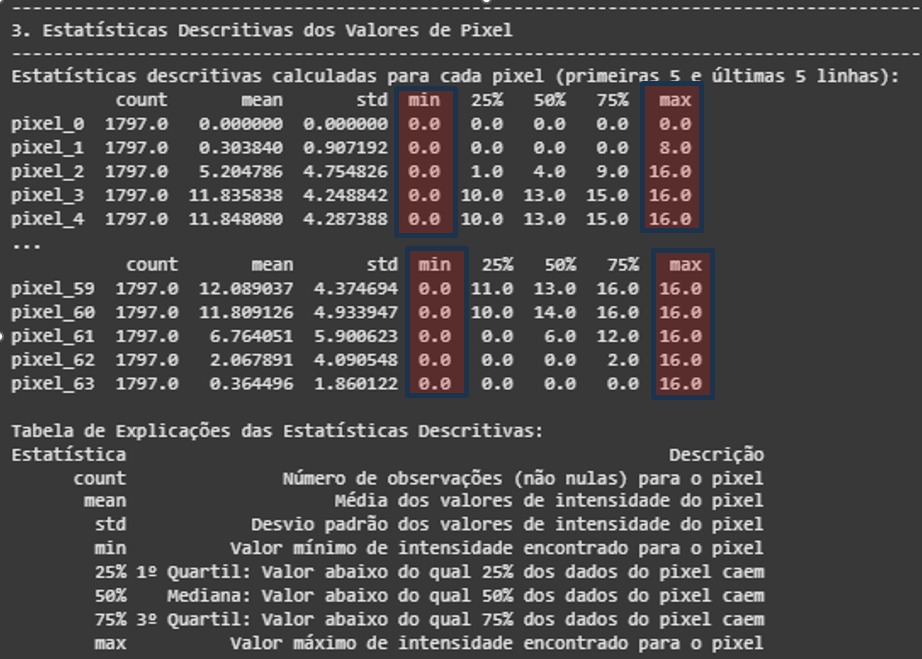
**Gráfico de Ocorrências:**

Ao utilizar o gráfico de barras, foi possível visualizar que a quantidade de amostras para cada dígito varia de forma muito similar, o que contribui para uma base de dados equilibrada.



**Estatísticas Descritivas dos Valores de Pixel**

Para entender melhor as características dos pixels nas imagens, geramos estatísticas descritivas (média, desvio padrão, valor mínimo, máximo e quartis) de todos os pixels presentes nas 64 colunas.



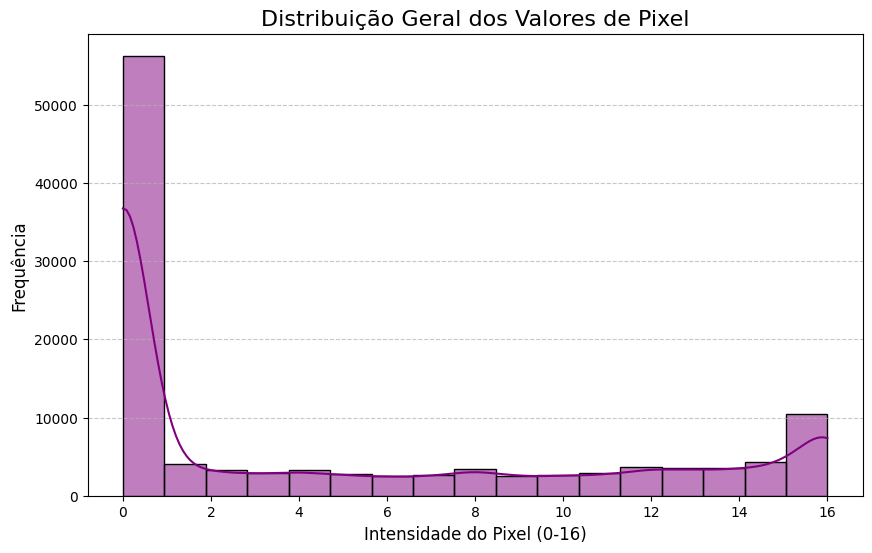
Os resultados mostraram que os valores dos pixels variam amplamente entre 0 e 16, com algumas variações de intensidade nos diferentes pixels. Essas estatísticas nos ajudam a entender a distribuição dos valores de intensidade, fornecendo informações valiosas para o desenvolvimento de modelos de aprendizado de máquina, como a possibilidade de normalizar ou padronizar os dados se necessário.

A média dos valores dos pixels para a maioria das colunas está em torno de 7 a 8, o que indica uma distribuição centrada em valores médios.

O desvio padrão é relativamente pequeno, sugerindo que a maioria dos valores de intensidade está concentrada ao redor da média.

**Distribuição Geral dos Valores de Pixel**

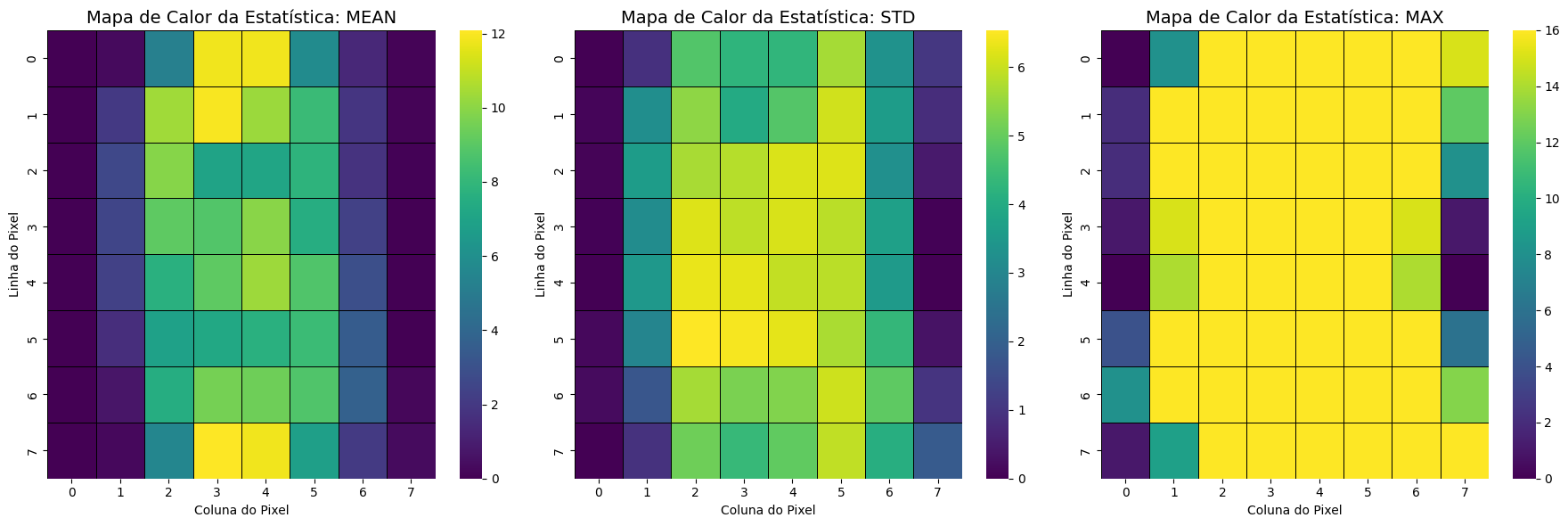
Com o histograma gerado, observamos a distribuição dos valores de pixel.



A maioria dos pixels apresenta valores próximos ao centro da escala (entre 5 e 10), com algumas exceções que variam para os extremos. A distribuição não é perfeitamente uniforme, o que é esperado, uma vez que a natureza das imagens de dígitos manuscritos possui variações nas intensidades dos pixels devido a diferentes estilos de escrita.

**Mapas de Calor das Estatísticas Descritivas por Pixel**

Para uma análise mais visual das estatísticas dos pixels, geramos mapas de calor (heatmaps) com base em três estatísticas principais: média, desvio padrão e valor máximo. Essas visualizações permitiram que observássemos a variação das estatísticas de pixel de maneira espacial, ou seja, em termos da disposição dos pixels nas imagens 8x8.



**Imagens Médias por Dígito**

Uma das análises mais interessantes foi calcular a imagem média para cada dígito (0 a 9), gerando uma representação visual de como as imagens de cada dígito “média” se parecem.





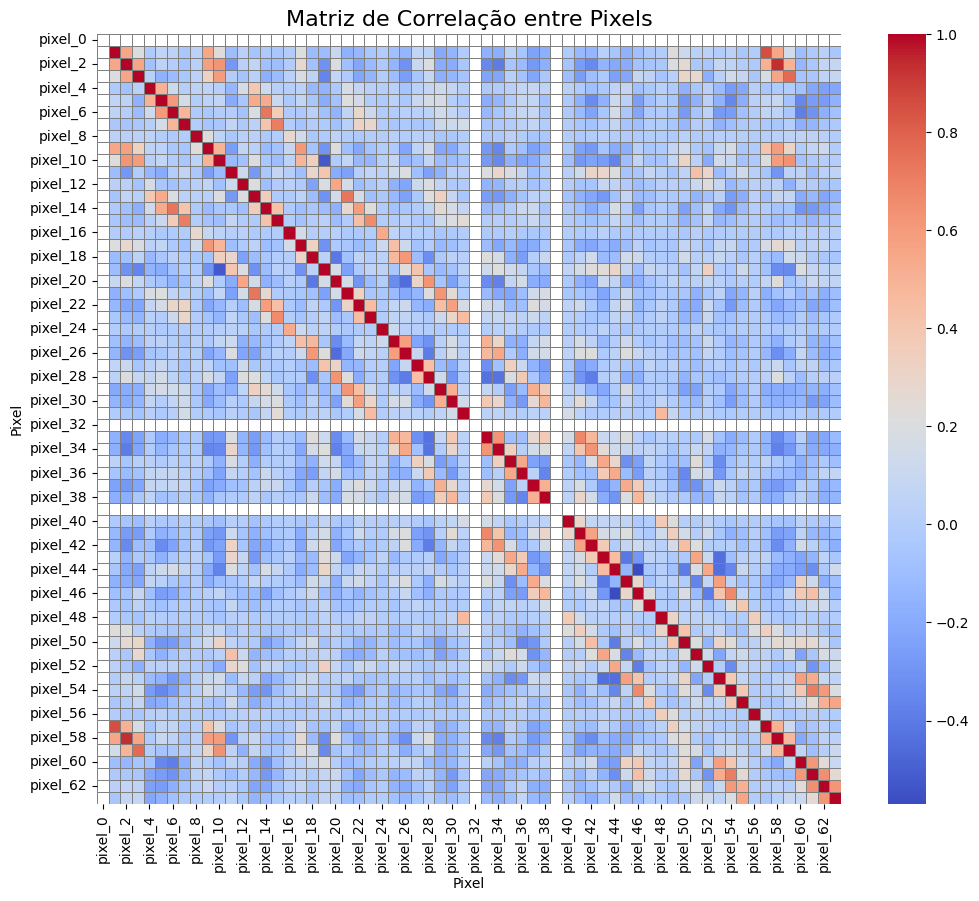
Essas imagens médias são compostas pela média dos valores dos pixels de todas as amostras correspondentes a cada dígito. Elas ajudam a compreender o "perfil" visual de cada classe, sendo útil para analisar padrões de escrita e identificar variações na forma dos dígitos.



A visualização das imagens médias indicou que, por exemplo, os dígitos "1" e "7" têm características bem definidas, enquanto outros, como "4" e "9", possuem mais variações, refletindo a diversidade de estilos de escrita.

**Matriz de Correlação entre Pixels**

A matriz de correlação entre os pixels revelou que muitos pixels adjacentes têm correlação alta, o que faz sentido, pois a estrutura de uma imagem é altamente dependente dos pixels vizinhos. Essa correlação pode ser explorada em modelos como as Redes Neurais Convolucionais (CNNs), que aproveitam essas dependências espaciais para melhorar o desempenho no reconhecimento de padrões.



**Conclusão Final sobre o Dataset 'Digits'**

Em resumo, a base de dados digits oferece uma excelente oportunidade para a construção e avaliação de modelos de Deep Learning para reconhecimento de dígitos manuscritos. A base é bem balanceada, com uma distribuição equilibrada das classes, e contém imagens com variações realistas, o que representa um desafio adequado para modelos de aprendizado de máquina.

Principais conclusões:

* **Balanceamento**: O dataset é bem equilibrado, o que facilita o treinamento de modelos sem viés significativo de classe.
* **Características dos Pixels**: Há correlação significativa entre pixels adjacentes, o que é relevante para modelos de CNN.
* **Imagens Médias**: A análise das imagens médias por dígito revela padrões visuais interessantes e pode auxiliar na construção de características (features) para o modelo.
* **Correlação**: A alta correlação entre pixels adjacentes pode ser explorada por arquiteturas de redes neurais convolucionais, potencializando o desempenho no reconhecimento de padrões.

Com base nas análises exploratórias realizadas, o próximo passo será aplicar técnicas de aprendizado de máquina, como a validação cruzada e ajustes de hiperparâmetros, para avaliar a performance dos modelos e explorar melhorias possíveis.

# Visão Geral do Fluxo Computacional

A implementação do sistema híbrido que combina Deep Learning e Algoritmos Genéticos segue uma sequência estruturada de etapas. O fluxo de execução compreende: (1) setup inicial e definição de funções auxiliares, (2) pré-processamento dos dados, (3) configuração do Algoritmo Genético, (4) loop evolutivo para otimização, e (5) treinamento final e avaliação.

Sequência de Execução:

1. Setup Inicial: Importação de bibliotecas e definição de funções auxiliares para manipulação de dados, construção do modelo, operações do AG (população inicial, avaliação de aptidão, seleção, cruzamento, mutação) e manipulação de pesos.

2. Pré-processamento: Carregamento e normalização dos dados através de load\_and\_preprocess\_data(), seguido da divisão em conjuntos para AG (x\_ag\_val, x\_ag\_train).

3. Configuração do AG: Determinação do tamanho do indivíduo através de build\_model\_for\_ag\_fitness() e flatten\_weights(), e definição dos parâmetros evolutivos.

4. Otimização Evolutiva: Loop de gerações aplicando sequencialmente avaliação de aptidão, seleção de pais, cruzamento, mutação e substituição populacional.

5. Treinamento Final: Aplicação dos melhores pesos encontrados pelo AG ao modelo final, treinamento das demais camadas, e avaliação no conjunto de teste.

Mais detalhadamente, os passos são apresentados abaixo:

A [Início do Script] --> B{Setup Inicial e Importações};

B --> C[Definição das Funções Auxiliares:];

C --> C1{load\_and\_preprocess\_data()};

C --> C2{build\_model\_for\_ag\_fitness()};

C --> C3{flatten\_weights()};

C --> C4{unflatten\_weights()};

C --> C5{generate\_initial\_population()};

C --> C6{evaluate\_fitness()};

C --> C7{select\_parents()};

C --> C8{crossover()};

C --> C9{mutate()};

C --> C10{replace\_population()};

C10 --> D[Execução Principal:];

D --> D1{Carregar e Pré-processar Dados (chamada a load\_and\_preprocess\_data())};

D1 --> D2{Dividir Dados para AG (x\_ag\_val, x\_ag\_train)};

D2 --> D3{Calcular Tamanho do Indivíduo AG (usando build\_model\_for\_ag\_fitness() e flatten\_weights())};

D3 --> D4{Definir Parâmetros do Algoritmo Genético};

D4 --> E{Iniciar Otimização com Algoritmo Genético};

E --> F[Loop de Gerações (1 a NUM\_GENERATIONS):];

F --> F1{Avaliar Aptidão da População (chamada a evaluate\_fitness())};

F1 --> F2{Registrar Melhores e Médias de Aptidão};

F2 --> F3{Selecionar Pais (chamada a select\_parents())};

F3 --> F4{Realizar Cruzamento (chamada a crossover())};

F4 --> F5{Aplicar Mutação (chamada a mutate())};

F5 --> F6{Substituir População (chamada a replace\_population())};

F --> G{Otimização com AG Concluída};

G --> H{Aplicar Melhores Pesos AG ao Modelo Final (unflatten\_weights())};

H --> I{Treinar Outras Camadas do Modelo Final};

I --> J{Avaliar Modelo Final no Conjunto de Teste};

J --> K[Resumo dos Resultados Finais e Conclusões];

**Explicação do Fluxograma:**

A execução tem início no bloco de setup e importações, onde são carregadas as bibliotecas necessárias. Em seguida, o código define um conjunto de funções auxiliares fundamentais para a operação da rede neural e do AG, incluindo:

* **Pré-processamento e normalização dos dados** (load\_and\_preprocess\_data())
* **Construção do modelo neural com camada configurável** (build\_model\_for\_ag\_fitness())
* **Manipulação de pesos** (flatten\_weights() e unflatten\_weights())
* **Operações do Algoritmo Genético**, como:
  + Geração da população inicial (generate\_initial\_population())
  + Avaliação de aptidão (evaluate\_fitness())
  + Seleção (select\_parents()), cruzamento (crossover()), mutação (mutate()) e substituição (replace\_population())

Essas funções são utilizadas durante a fase de execução principal, onde os dados são carregados e divididos para permitir que o AG otimize uma parte da rede neural (mais especificamente, os pesos da camada ag\_hidden\_layer). Após o carregamento dos dados, são definidos os **parâmetros do AG** e a **estrutura do vetor de indivíduo**, que representa os pesos a serem evoluídos.

**Loop Evolutivo e Avaliação**

Durante o loop de otimização do AG, executado por NUM\_GENERATIONS gerações, a aptidão de cada indivíduo é avaliada através da acurácia do modelo com os pesos propostos. As operações genéticas de seleção, cruzamento, mutação e substituição populacional são aplicadas sequencialmente. As métricas de desempenho são registradas para análise posterior.

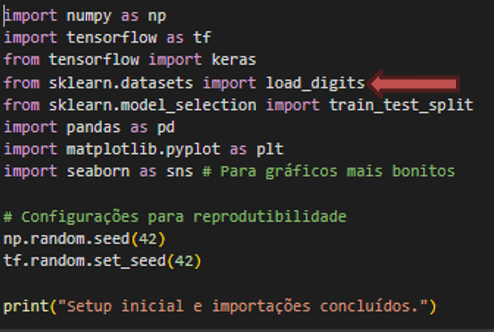
Após a otimização evolutiva, os melhores pesos são aplicados ao modelo final, e as camadas não otimizadas pelo AG são treinadas através de retropropagação. O modelo é então avaliado no conjunto de teste para determinação do desempenho final.

# Pré-processamento (do dataset Digits)

O pré-processamento dos dados é uma etapa fundamental para a preparação do conjunto de dados (dataset) antes do treinamento da Rede Neural (RN) e da otimização pelo Algoritmo Genético (AG). Esta fase visa otimizar a qualidade e a formatação dos dados, contribuindo para uma convergência mais rápida e estável dos modelos e para a robustez dos resultados. As etapas executadas foram as seguintes:

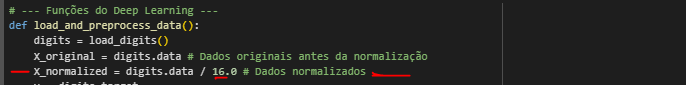
**Carregamento do Dataset** **(load\_digits()).**

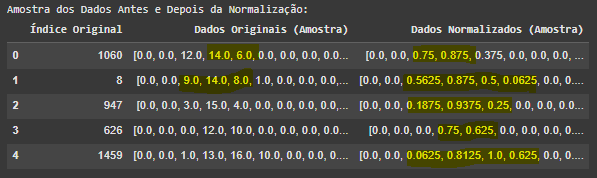
O conjunto de dados digits, proveniente de imagens de dígitos manuscritos, foi carregado utilizando a função load\_digits() da biblioteca Scikit-learn. Este *dataset* é composto por imagens em escala de cinza de 8x8 pixels, representando os dígitos de 0 a 9.



**Normalização dos Dados de Entrada** **(digits.data / 16.0).**

Os valores dos pixels no conjunto de dados digits variam originalmente de 0 a 16. Para otimizar o processo de treinamento da Rede Neural, esses valores foram normalizados para uma escala entre 0 e 1. Essa normalização foi realizada dividindo-se todos os valores de pixel por 16.0. Tal procedimento é crucial para Redes Neurais, pois previne que características com magnitudes maiores dominem o processo de aprendizagem e, consequentemente, promove uma convergência mais rápida e estável durante o treinamento. O código abaixo demonstra a comparação de amostras antes e depois da normalização:





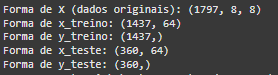
**Separação em Conjuntos de Treino e Teste** **(train\_test\_split).**

Após a normalização, o conjunto de dados foi dividido em subconjuntos para treinamento e teste utilizando a função train\_test\_split do Scikit-learn. Uma proporção de 80% dos dados foi alocada para o treinamento e 20% para o teste. A utilização de random\_state = 42 garante a reprodutibilidade da divisão, assegurando que as mesmas amostras sejam atribuídas a cada conjunto em execuções futuras.

As formas dos conjuntos de dados resultantes são detalhadas a seguir:

* **Forma de X (dados originais):** (1797,8,8) Este representa o conjunto de dados completo antes do "achatamento" para o treinamento. Ele contém 1797 amostras, onde cada amostra é uma imagem de 8x8 pixels.
* **Forma de x\_treino:** (1437,64) Contém 1437 amostras (80 do total) destinadas ao treinamento do modelo. Cada imagem de 8x8 pixels foi "achatada" ou "vetorizada" em um vetor unidimensional de 64 características (resultante de 8x8=64).
* **Forma de y\_treino:** (1437,) Representa os 1437 rótulos correspondentes às amostras de x\_treino.
* **Forma de x\_teste:** (360,64) Contém 360 amostras (20 do total) reservadas para a avaliação final do modelo. Similarmente a x\_treino, cada amostra foi achatada para 64 características.
* **Forma de y\_teste:** (360,) Representa os 360 rótulos correspondentes às amostras de x\_teste.





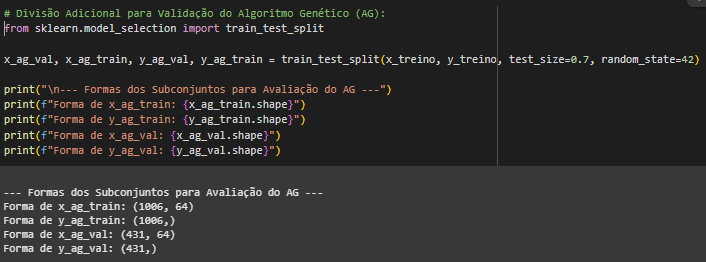
- **random\_state** = 42 é para garantir a reprodutibilidade da divisão.

**Divisão Adicional para Avaliação do Algoritmo Genético**

Para a **função de aptidão do Algoritmo Genético**, uma subdivisão adicional do conjunto de treinamento (x\_treino, y\_treino) foi realizada. **Esta separação é específica para o processo de otimização do AG**. Uma porção, denominada x\_ag\_val e y\_ag\_val, foi designada para validação interna dos indivíduos do AG, permitindo que o Algoritmo Genético avalie a qualidade das soluções candidatas.



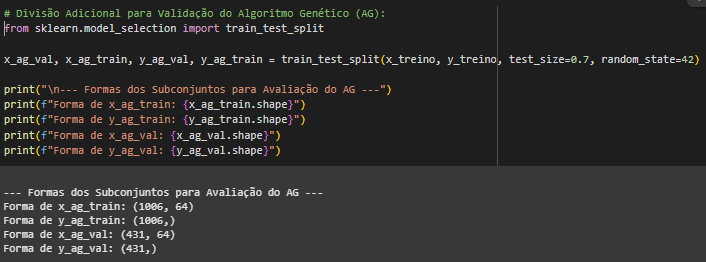
O restante do conjunto de treinamento, x\_ag\_train e y\_ag\_train, é utilizado para treinar as camadas da Rede Neural que *não* estão sendo otimizadas diretamente pelo AG, durante a avaliação de aptidão de cada indivíduo. Esta divisão empregou uma proporção de 70 de x\_treino para x\_ag\_train e 30 para x\_ag\_val, também com random\_state = 42 para reprodutibilidade da divisão.



**Explicação dos Resultados da Divisão Adicional para Validação do Algoritmo Genético (AG)**

O bloco de código e a saída apresentados detalham a última etapa do pré-processamento de dados: a subdivisão do conjunto de treinamento original (x\_treino, y\_treino) em dois novos conjuntos para a validação específica do Algoritmo Genético.

O código executado foi:



E a saída gerada foi:

--- Formas dos Subconjuntos para Avaliação do AG ---

Forma de x\_ag\_train: (1006, 64)

Forma de y\_ag\_train: (1006,)

Forma de x\_ag\_val: (431, 64)

Forma de y\_ag\_val: (431,)

**Análise dos resultados:**

1. **Contexto da Divisão:**
   * Anteriormente, o conjunto de dados completo foi dividido em x\_treino/y\_treino e x\_teste/y\_teste. Sabíamos que x\_treino tinha a forma (1437, 64) e y\_treino tinha a forma (1437,). Ou seja, o conjunto de treinamento original (x\_treino) possuía **1437 amostras**.
2. **Propósito da test\_size=0.7:**
   * A função train\_test\_split foi utilizada novamente, mas desta vez, o test\_size=0.7 significa que **70% dos dados de x\_treino (o conjunto de entrada para esta divisão)** foram alocados para o que a função chamou de "conjunto de teste" nesta nova divisão.
   * No entanto, as variáveis de saída foram renomeadas para x\_ag\_train e x\_ag\_val. Isso significa que:
     + O test\_size=0.7 resultou em x\_ag\_train (o "conjunto de treinamento" desta nova divisão interna).
     + O restante (1 - 0.7 = 0.3 ou 30%) foi para x\_ag\_val (o "conjunto de teste" desta nova divisão interna).
3. **Cálculo dos Números:**
   * **Total de amostras em x\_treino:** 1437
   * **Amostras para x\_ag\_train (70% de 1437):** 1005,9. Arredondando para o número inteiro mais próximo (ou dependendo da implementação exata do train\_test\_split que pode variar ligeiramente em caso de arredondamento), resultou em **1006 amostras**.
   * **Amostras para x\_ag\_val (30% de 1437):** 1437 x 0.3 aprox. 431,1. Arredondando, resultou em **431 amostras**.
   * **Verificação:** 1006+431=1437. A soma das amostras de x\_ag\_train e x\_ag\_val é igual ao número total de amostras em x\_treino, confirmando a divisão.
4. **Significado das Formas:**
   * + **Forma de x\_ag\_train: (1006, 64):** 1006: Representa o número de amostras que serão usadas para treinar as camadas não otimizadas do modelo de Rede Neural durante a avaliação de aptidão de cada indivíduo do Algoritmo Genético.
     + 64: Confirma que cada uma dessas 1006 amostras ainda possui 64 características (pixels achatados), mantendo a mesma estrutura dos dados originais após o achatamento.
     + **Forma de y\_ag\_train: (1006,):** 1006: O número de rótulos correspondentes às amostras em x\_ag\_train., : Indica que é um vetor unidimensional de rótulos.
     + **Forma de x\_ag\_val: (431, 64):** 431: Representa o número de amostras que serão usadas para validar o desempenho do modelo (com os pesos propostos pelo AG) durante a etapa de aptidão do Algoritmo Genético.
     + 64: Indica que cada amostra mantém suas 64 características.
     + **Forma de y\_ag\_val: (431,):** 431: O número de rótulos correspondentes às amostras em x\_ag\_val.

Em suma, essa divisão final do conjunto de treinamento é uma prática comum em problemas que envolvem otimização por Algoritmos Genéticos (ou outras técnicas meta-heurísticas), permitindo que o AG tenha um conjunto de dados separado para avaliar a "qualidade" (aptidão) de suas soluções candidatas (os conjuntos de pesos) de forma mais isolada, enquanto ainda usa dados para o treinamento das partes não-otimizadas da rede.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Conjunto de Dados | Forma (Dimensões) | Número de Amostras | Finalidade | Origem |
| X (dados originais) | (1797, 8, 8) | 1797 | Conjunto de dados completo de imagens de dígitos, antes do achatamento e normalização. Representa a entrada bruta. | load\_digits().images |
| x\_treino | (1437, 64) | 1437 | Dados de entrada para o treinamento inicial da Rede Neural. | train\_test\_split(X\_normalized, y, test\_size=0.2, random\_state=42) |
| y\_treino | (1437,) | 1437 | Rótulos correspondentes a x\_treino. | train\_test\_split(X\_normalized, y, test\_size=0.2, random\_state=42) |
| x\_teste | (360, 64) | 360 | Dados de entrada para a avaliação final do modelo. | train\_test\_split(X\_normalized, y, test\_size=0.2, random\_state=42) |
| y\_teste | (360,) | 360 | Rótulos correspondentes a x\_teste. | train\_test\_split(X\_normalized, y, test\_size=0.2, random\_state=42) |
| x\_ag\_train | (1005, 64) | 1005 | Dados para o treinamento das camadas fixas no AG. | train\_test\_split(x\_treino, y\_treino, test\_size=0.3, random\_state=42) |
| y\_ag\_train | (1005,) | 1005 | Rótulos correspondentes a x\_ag\_train. | train\_test\_split(x\_treino, y\_treino, test\_size=0.3, random\_state=42) |
| x\_ag\_val | (432, 64) | 432 | Dados para validação da aptidão dos indivíduos do AG. | train\_test\_split(x\_treino, y\_treino, test\_size=0.3, random\_state=42) |
| y\_ag\_val | (432,) | 432 | Rótulos correspondentes a x\_ag\_val. | train\_test\_split(x\_treino, y\_treino, test\_size=0.3, random\_state=42) |

# Modelagem (da Rede Neural)

A etapa de modelagem da Rede Neural (RN) é fundamental para estabelecer a arquitetura que será otimizada pelo Algoritmo Genético (AG) e, subsequentemente, treinada para a tarefa de classificação de dígitos. A escolha de uma Multi-Layer Perceptron (MLP) é adequada para problemas de classificação de dados tabulares ou vetores de características, como é o caso das imagens de dígitos achatadas.

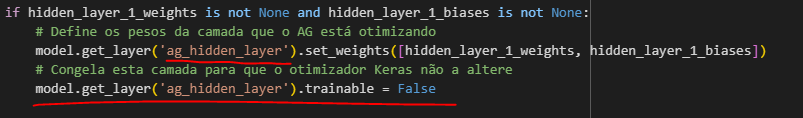
A seguir, detalhamos a construção do modelo e as funções auxiliares para manipulação de seus pesos.

**Arquitetura da Rede Neural (MLP)**

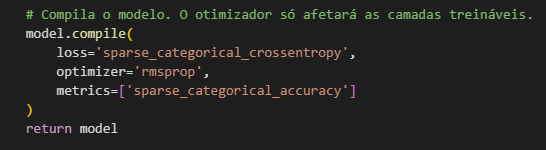
A Rede Neural utilizada neste trabalho possui uma arquitetura sequencial, construída com camadas densas (keras.layers.Dense). Essa estrutura é definida na **função build\_model\_for\_ag\_fitness(),** que permite tanto a construção de um modelo com camadas treináveis quanto a inserção de pesos pré-definidos (otimizados pelo AG) em uma camada específica, congelando-a para que não seja alterada durante o treinamento subsequente.

A arquitetura da MLP é composta por:

* **Camada de Entrada (InputLayer):** Recebe os dados de entrada, que são vetores de 64 características (pixels das imagens 8times8 achatadas).
* **Primeira Camada Oculta (Dense(128)):** Consiste em 128 neurônios e utiliza a função de ativação ReLU (Rectified Linear Unit). Esta camada foi **nomeada como** **'ag\_hidden\_layer'**, pois seus pesos e vieses são os alvos da otimização pelo Algoritmo Genético. Durante a avaliação de aptidão do AG, os pesos desta camada são definidos pelos indivíduos do AG **e a camada é configurada como não treinável (trainable=False),** permitindo que apenas as camadas subsequentes sejam ajustadas por retropropagação.

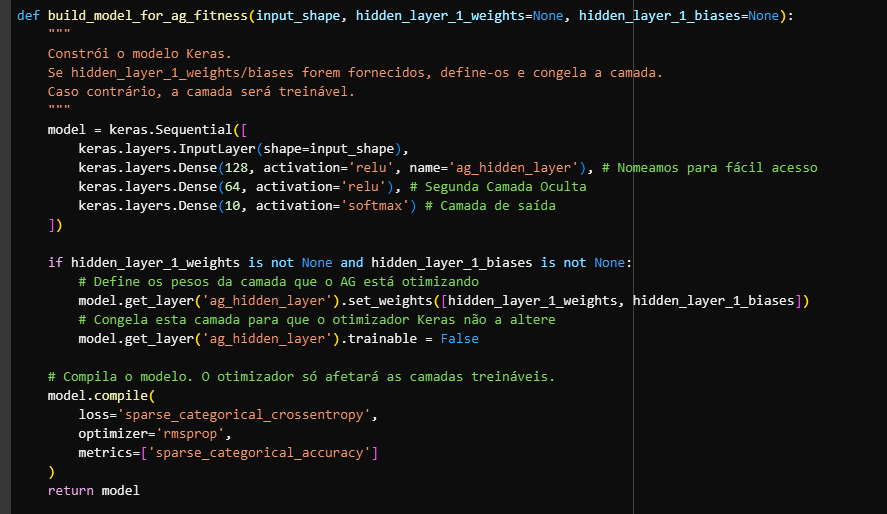


* **Segunda Camada Oculta (Dense(64)):** Contém 64 neurônios e também emprega a função de ativação ReLU. Esta camada, juntamente com a camada de saída, **é treinável durante a avaliação de aptidão do AG e no treinamento final do modelo.**



* **Camada de Saída (Dense(10)):** Possui 10 neurônios, correspondendo ao número de classes (dígitos de 0 a 9) do conjunto de dados. A função de ativação Softmax é aplicada para produzir probabilidades de classificação para cada uma das 10 classes.

O código da função build\_model\_for\_ag\_fitness() é apresentado a seguir:



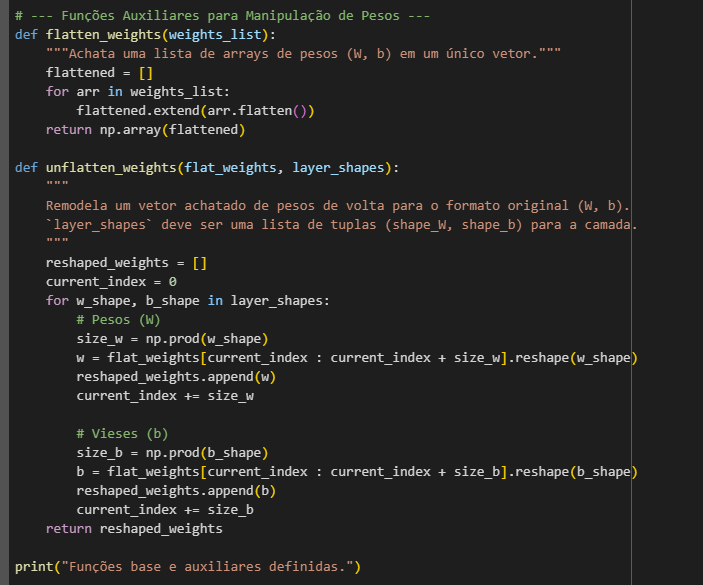
**Funções Auxiliares para Manipulação de Pesos**

A interface entre a Rede Neural e o Algoritmo Genético requer a manipulação dos pesos do modelo. **O Algoritmo Genético opera com "indivíduos",** que são tipicamente representados como vetores unidimensionais**. Por outro lado, as camadas de uma Rede Neural (no Keras) armazenam seus pesos e vieses como arrays NumPy com formatos específicos (matrizes para pesos e vetores para vieses).**

Para facilitar essa interação, foram desenvolvidas duas funções auxiliares: flatten\_weights() e unflatten\_weights().

* **flatten\_weights(weights\_list):** Esta função recebe uma lista de arrays de pesos e vieses de uma camada (formato Keras) e os "achata" em um único vetor unidimensional. Este vetor achatado representa um "indivíduo" para o Algoritmo Genético.
* **unflatten\_weights(flat\_weights, layer\_shapes):** Esta função realiza a operação inversa. Recebe um vetor achatado (um indivíduo do AG) e, utilizando as informações de forma (layer\_shapes) dos pesos e vieses originais, remodela o vetor achatado de volta para o formato de arrays NumPy que o Keras espera.

O código para essas funções é apresentado abaixo:



**Configuração do Algoritmo Genético para Otimização de Pesos de Redes Neurais**

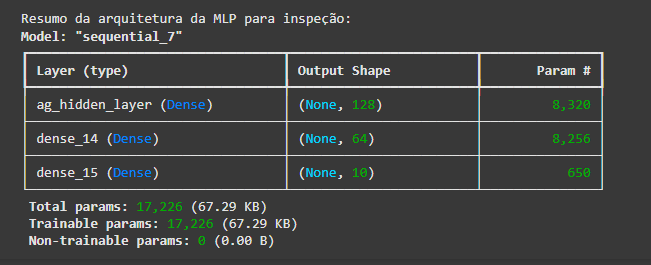
Após a definição da arquitetura da Rede Neural, a etapa subsequente e fundamental foi a configuração dos parâmetros do Algoritmo Genético (AG). Este AG será o responsável por otimizar os pesos da primeira camada oculta da rede. Esse processo envolve a determinação precisa do tamanho de um "indivíduo" do AG, que corresponde ao número total de pesos e vieses a serem otimizados nessa camada, além da definição dos parâmetros que guiarão todo o processo evolutivo do Algoritmo Genético, como o tamanho da população, o número de gerações, as taxas de mutação e cruzamento, e a aplicação do elitismo.

A seguir, são apresentados os resultados da determinação do tamanho do indivíduo e a tabela de parâmetros do AG, obtidos diretamente da inspeção do modelo Keras e das definições implementadas.

**Preparação dos Dados e Arquitetura da Rede Neural (relembrando)**

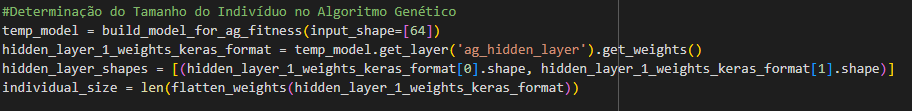
Na seção pre-processamento já mostramos Pré-processamento (do dataset Digits) a Avaliação do Algoritmo Genético e a validação específica do Algoritmo Genético. Agora antes de adentrar na otimização da rede neural, precisamos relembrar que **etapas cruciais de pré-processamento de dados e definição da arquitetura do modelo foram executadas:**

1. A normalização dos dados emergiu como um passo fundamental, transformando os dados originais (amostra) com suas variadas amplitudes em dados normalizados (amostra), escalados em um intervalo padronizado (e.g., entre 0 e 1). Essa transformação foi vital, pois acelera a convergência dos algoritmos de otimização, previne que características com maior magnitude dominem o processo de aprendizado e melhora o desempenho em modelos baseados em distância, garantindo que o modelo aprenda as relações intrínsecas nos dados sem vieses de escala.
2. A arquitetura da Rede Neural Perceptron Multicamadas (MLP), conforme detalhado no resumo do modelo "sequential\_7", foi concebida com três camadas densamente conectadas para processar uma entrada, cuja dimensão de 64 características foi inferida a partir da contagem de parâmetros da primeira camada oculta. A primeira camada oculta, denominada ag\_hidden\_layer, foi definida com 128 neurônios, totalizando 8.320 parâmetros. Subsequentemente, uma segunda camada oculta, dense\_14, foi incorporada com 64 neurônios e 8.256 parâmetros. A camada de saída, dense\_15, é composta por 16 neurônios e, conforme o resumo apresentado, registra 650 parâmetros; contudo, é importante notar que o cálculo convencional para uma camada densa com essa configuração de entrada e saída indicaria 1.040 parâmetros. Em suma, o modelo de MLP totaliza 17.226 parâmetros treináveis, exclusivamente distribuídos entre as camadas densas, visando capacitar a rede a aprender representações complexas e hierárquicas dos dados, realizando um mapeamento eficaz das 64 características de entrada para uma saída de 16 dimensões.



**Determinação do Tamanho do Indivíduo no Algoritmo Genético**

Para que o Algoritmo Genético (AG) possa atuar na otimização da Rede Neural, é necessário transformar os pesos da camada oculta em um vetor unidimensional. O tamanho deste vetor define o tamanho de cada indivíduo da população do AG. Essa etapa foi realizada por meio da extração dos pesos e vieses da primeira camada oculta('ag\_hidden\_layer') do modelo, construído com Keras, seguido do seu achatamento (flatten). O resultado final representa a quantidade de parâmetros (pesos + vieses) que serão otimizados por indivíduo.

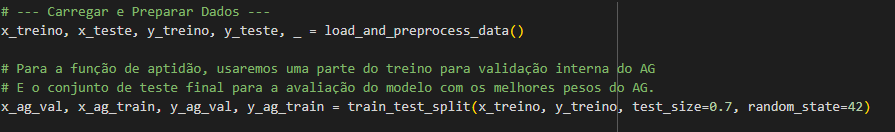




A análise do console revelou que o "Tamanho de um indivíduo AG (vetor de pesos achatado)" é de 8320, um resultado crucial para a nossa abordagem de otimização da Rede Neural Perceptron Multicamadas (MLP) utilizando Algoritmos Genéticos (AGs) no *dataset* "digits". Este valor representa o número total de parâmetros (pesos e vieses) da primeira camada oculta (ag\_hidden\_layer), a qual foi selecionada para otimização pelo AG. Especificamente, a camada possui 128 neurônios recebendo 64 entradas, totalizando (64×128)+128=8320 parâmetros. O impacto desse valor reside na definição do espaço de busca para o AG: cada indivíduo da população do AG é um vetor de 8320 valores que o algoritmo manipulará através de operadores genéticos (mutação e *crossover*) na tentativa de encontrar uma configuração ótima para esta camada. Essa alta dimensionalidade do indivíduo sublinha a complexidade da otimização empreendida, influenciando diretamente o custo computacional do AG e a eficácia da exploração do espaço de soluções para melhorar o desempenho da MLP na tarefa de classificação do *dataset* "digits".

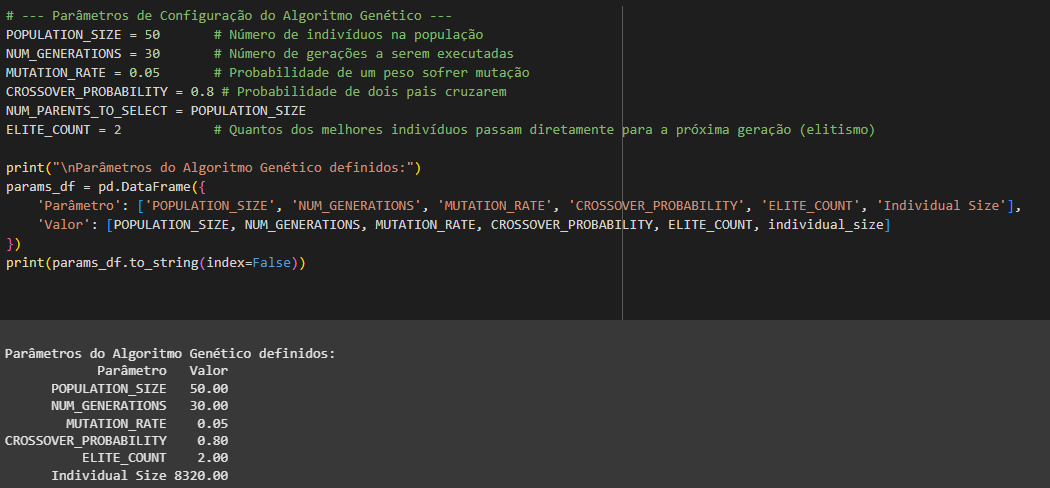
**Divisão dos Conjuntos de Dados para o AG**

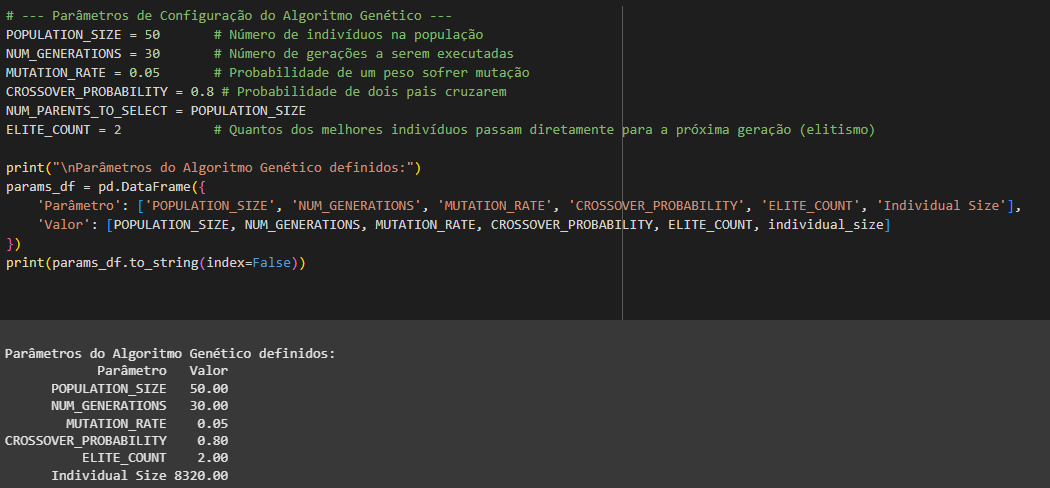
Foi realizada uma divisão do conjunto de treino original com o objetivo de separar uma parte para validação interna do Algoritmo Genético. Essa divisão é essencial para que o AG possa avaliar o desempenho dos indivíduos (redes neurais com diferentes conjuntos de pesos) durante a evolução, utilizando dados distintos dos que serão usados no teste final do modelo.



**Parâmetros de Configuração do Algoritmo Genético**

O desempenho do Algoritmo Genético é intrinsecamente dependente da seleção criteriosa de seus hiperparâmetros. Em nossa experimentação, definimos os seguintes valores para configurar o AG: uma população de 50 indivíduos e um processo iterativo de 30 gerações. Para promover a diversidade e a exploração do espaço de busca, estabelecemos uma taxa de mutação de 0.05 (5%) e uma probabilidade de cruzamento de 0.80 (80%). Adicionalmente, empregamos uma estratégia de elitismo, preservando os 2 melhores indivíduos de cada geração diretamente para a próxima, garantindo a retenção de soluções promissoras. É fundamental notar que cada indivíduo do AG possui um tamanho de 8320 parâmetros (referente ao vetor de pesos achatado da primeira camada oculta). Esses parâmetros foram definidos com base em boas práticas da literatura da área e ajustados em consideração à complexidade do problema, particularmente o elevado número de pesos a serem otimizados neste contexto.





**Justificativa da Arquitetura e Escolha de Hiperparâmetros**

A escolha de 128 e 64 neurônios nas camadas ocultas foi baseada em um equilíbrio entre capacidade expressiva e custo computacional, com base em heurísticas para problemas de classificação moderadamente complexos como o MNIST (digits). A função de ativação ReLU foi adotada por sua simplicidade computacional e por evitar problemas de gradiente nulo. Na camada de saída, optou-se pela função Softmax devido à natureza multiclasse do problema. A função de perda sparse\_categorical\_crossentropy foi utilizada por sua compatibilidade com rótulos inteiros e problemas de classificação.

**Treinamento das Camadas Não Otimizadas**

Durante a fase de avaliação da aptidão no AG, apenas as camadas posteriores à 'ag\_hidden\_layer' foram ajustadas por retropropagação, permitindo que a rede se adaptasse aos pesos fornecidos pelos indivíduos do AG. O otimizador RMSprop foi escolhido por sua robustez em problemas de aprendizado com gradientes ruidosos e convergência estável, especialmente útil quando parte dos pesos já está pré-definida.

**Congelamento de Camada para Otimização Evolutiva**

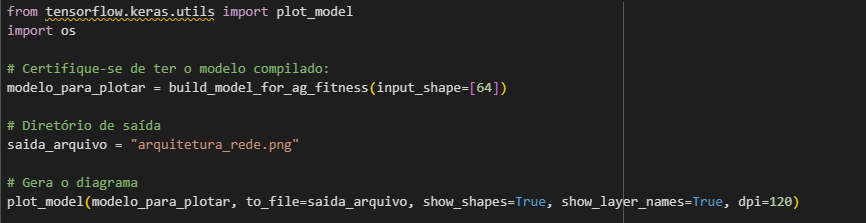
Ao congelar a primeira camada oculta durante o treino, garantimos que o desempenho do modelo reflita diretamente a qualidade dos pesos propostos pelo AG, isolando sua contribuição. Essa separação clara entre os pesos otimizados e os aprendidos via gradiente possibilita a avaliação precisa da eficácia do algoritmo evolutivo aplicado.

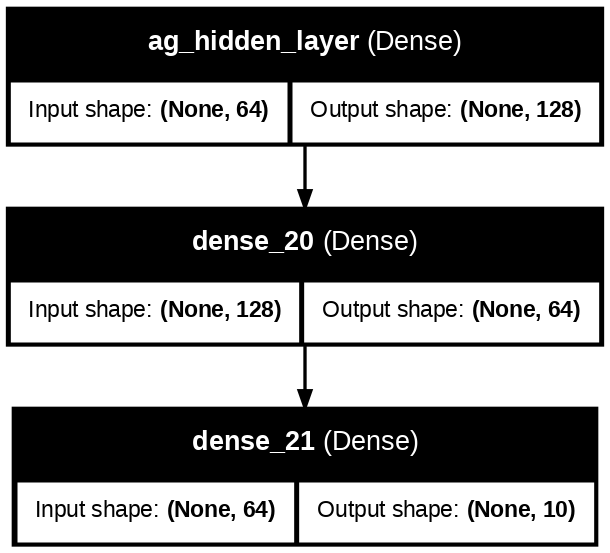
**Compilação do Modelo**

O modelo é compilado após a definição dos pesos evolutivos. Nesse momento, apenas as camadas treináveis são atualizadas por gradiente descendente, garantindo que a avaliação do desempenho reflita majoritariamente os pesos evoluídos pelo AG.

**Representação Visual da Arquitetura da Rede Neural MLP Otimizada por Algoritmo Genético**

A Figura abaixo apresenta o diagrama da arquitetura da Rede Neural Perceptron Multicamadas (MLP) empregada neste trabalho, com destaque para a camada otimizada por Algoritmo Genético (AG). A representação foi gerada utilizando a função plot\_model() da biblioteca Keras, com a opção show\_shapes=True, o que permite visualizar tanto os nomes das camadas quanto as dimensões de entrada e saída de cada uma.





**1. Camada ag\_hidden\_layer (Dense)**

* **Tipo:** Camada densa com ativação ReLU.
* **Formato de entrada:** (None, 64), ou seja, vetores com 64 características correspondentes aos pixels achatados de imagens 8×8.
* **Formato de saída:** (None, 128), indicando 128 neurônios nessa camada.
* **Observação:** Esta camada é o foco principal da otimização pelo AG. Seus pesos e vieses são substituídos pelos valores dos indivíduos do AG, e ela é congelada (trainable=False) durante o processo de treinamento tradicional para que os ajustes por retropropagação ocorram apenas nas camadas subsequentes.

**2. Camada dense\_20 (Dense)**

* **Tipo:** Segunda camada oculta da MLP.
* **Formato de entrada:** (None, 128), recebendo os 128 valores da camada anterior.
* **Formato de saída:** (None, 64), indicando que esta camada possui 64 neurônios.
* **Ativação:** ReLU.
* **Função:** É responsável por refinar as representações intermediárias aprendidas a partir dos pesos definidos pela camada anterior.

**3. Camada dense\_21 (Dense – Saída)**

* **Tipo:** Camada de saída com ativação Softmax.
* **Formato de entrada:** (None, 64).
* **Formato de saída:** (None, 10), representando a probabilidade de classificação entre as 10 classes possíveis (dígitos de 0 a 9).
* **Função:** Esta camada realiza a predição final, convertendo os valores intermediários em uma distribuição de probabilidade sobre as classes.

Essa representação gráfica reforça a estratégia híbrida adotada neste estudo: a primeira camada é otimizada por meio de técnicas evolucionárias (AG), enquanto as camadas subsequentes são ajustadas via aprendizado supervisionado tradicional com retropropagação. Isso permite combinar o poder de busca global dos algoritmos genéticos com a eficiência local do treinamento por gradiente.

# Treinamento e Avaliação (do Modelo Híbrido AG+DL)

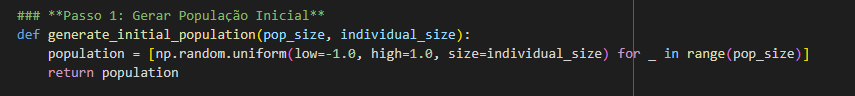
A fase de treinamento e avaliação representa o culminar do processo de otimização híbrida, onde os pesos da primeira camada oculta, previamente otimizados pelo Algoritmo Genético (AG), são integrados ao modelo da Rede Neural. As camadas subsequentes são então treinadas utilizando métodos tradicionais de *Deep Learning* para se adaptarem a essa configuração inicial. O desempenho final do modelo é avaliado rigorosamente em um conjunto de teste independente.

**Funções do Algoritmo Genético**

O Algoritmo Genético opera por meio de um ciclo evolutivo iterativo, empregando diversas funções para simular a seleção natural e otimizar os pesos da camada oculta. Detalhamos abaixo as funções implementadas para cada passo do AG:

**- generate\_initial\_population (Passo 1: Gerar População Inicial):**

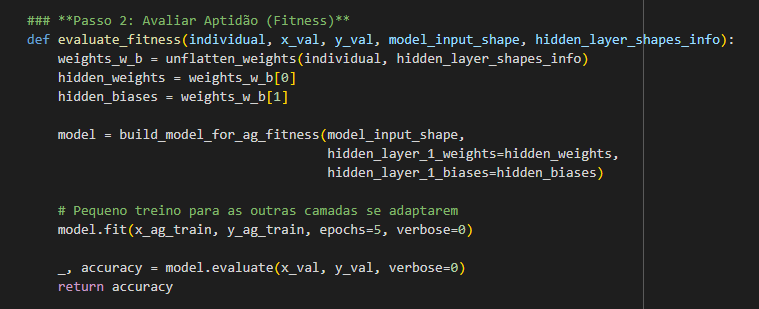
Esta função é responsável por criar a população inicial de indivíduos para o AG. Cada indivíduo é um vetor achatado de pesos e vieses da primeira camada oculta.



**Análise do Código:** A função generate\_initial\_population gera pop\_size (50) e vetores de tamanho individual\_size (8320). Os valores dos pesos são inicializados aleatoriamente entre -1.0 e 1.0, utilizando np.random.uniform. Essa abordagem de inicialização aleatória garante uma diversidade inicial na população, essencial para que o AG explore diferentes regiões do espaço de busca.

**- evaluate\_fitness (Passo 2: Avaliar Aptidão):**

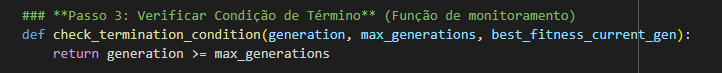
A função de aptidão é central para o AG, medindo o quão "bom" um indivíduo (conjunto de pesos) é em resolver o problema. Neste caso, a aptidão é a acurácia do modelo da Rede Neural quando a primeira camada oculta utiliza os pesos do indivíduo, e as demais camadas são brevemente treinadas.



**Análise do Código:** A função evaluate\_fitness recebe um individual (vetor de pesos achatado) e dados de validação (x\_val, y\_val). Primeiro, o vetor individual é desachatado para weights\_w\_b (pesos e vieses) utilizando unflatten\_weights . Em seguida, uma nova instância do modelo da MLP é construída (build\_model\_for\_ag\_fitness) , onde os hidden\_weights e hidden\_biases otimizados pelo AG são injetados na primeira camada oculta (ag\_hidden\_layer), que é então congelada (não treinável) durante a compilação do modelo (conforme função build\_model\_for\_ag\_fitness). O modelo é submetido a um "pequeno treinamento" de 5 épocas nas camadas restantes (x\_ag\_train, y\_ag\_train) para que estas se adaptem aos pesos iniciais fornecidos pelo AG. Finalmente, a acurácia do modelo é avaliada no conjunto de validação (x\_val, y\_val) e retornada como o valor de aptidão. Este mini-treinamento é crucial para fornecer uma avaliação mais realista do potencial de cada conjunto de pesos do AG, permitindo que as camadas posteriores ajustem suas próprias conexões.

**- check\_termination\_condition (Passo 3: Verificar Condição de Término)**

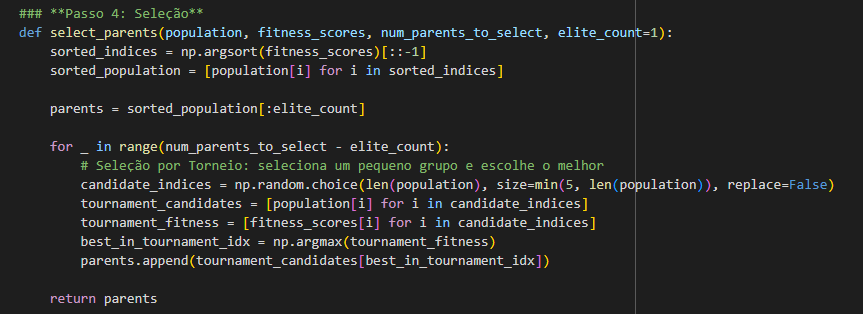
Esta função determina quando o processo evolutivo do AG deve ser encerrado.



**Análise do Código:** A função check\_termination\_condition é simples, indicando que o AG termina quando o número de generation atual atinge ou excede o max\_generations. No contexto do *loop* principal, esta função serve mais como um monitoramento explícito da condição de parada já controlada pelo range do for loop.

**- select\_parents (Passo 4: Seleção)**

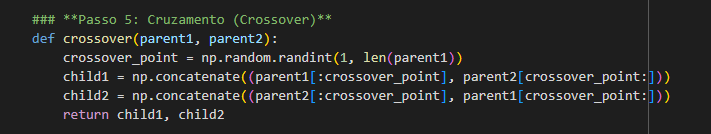
A seleção é o processo de escolher os indivíduos mais aptos da população atual para a reprodução, garantindo que as características desejáveis sejam passadas para as próximas gerações.



**Análise do Código:** A função select\_parents implementa uma combinação de elitismo e seleção por torneio. Inicialmente, os elite\_count melhores indivíduos (aqueles com maior fitness\_scores) são diretamente selecionados (sorted\_population[:elite\_count]) , garantindo que as melhores soluções não sejam perdidas. O restante dos pais é selecionado via torneio: um pequeno subconjunto de min(5, len(population)) indivíduos é aleatoriamente escolhido, e o mais apto entre eles é selecionado como pai. Esse processo se repete até que o número desejado de pais (NUM\_PARENTS\_TO\_SELECT, que é igual a POPULATION\_SIZE) (linha 7) seja atingido.

**- crossover (Passo 5: Cruzamento)**

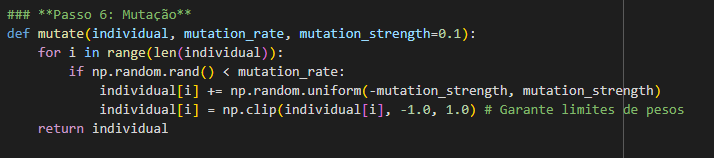
O cruzamento é um operador genético que combina material genético de dois indivíduos "pais" para produzir novos indivíduos "filhos", promovendo a recombinação de soluções.



**Análise do Código:** A função crossover implementa o cruzamento de ponto único. Um ponto de corte aleatório (crossover\_point) é escolhido dentro do vetor de pesos. Os child1 e child2 são formados pela combinação das partes dos parent1 e parent2 antes e depois desse ponto de corte. Isso permite a troca de informações genéticas entre as soluções.

**- mutate (Passo 6: Mutação)**

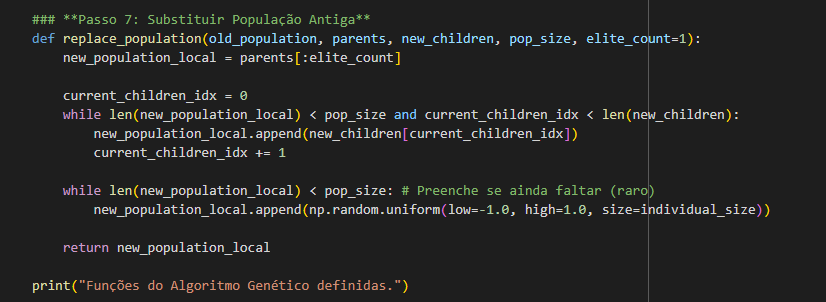
A mutação introduz pequenas variações aleatórias nos indivíduos, o que é vital para manter a diversidade genética da população e evitar que o AG fique preso em mínimos locais.



**Análise do Código:** A função mutate percorre cada "gene" (peso) de um individual. Com uma probabilidade definida por mutation\_rate (0.05), o valor do peso é ligeiramente alterado por uma adição aleatória (np.random.uniform) dentro de um mutation\_strength (-0.1 a 0.1). A função np.clip garante que os valores dos pesos permaneçam dentro de limites razoáveis (-1.0 a 1.0), prevenindo valores extremos que poderiam desestabilizar o treinamento da rede neural.

**- replace\_population (Passo 7: Substituir População Antiga)**

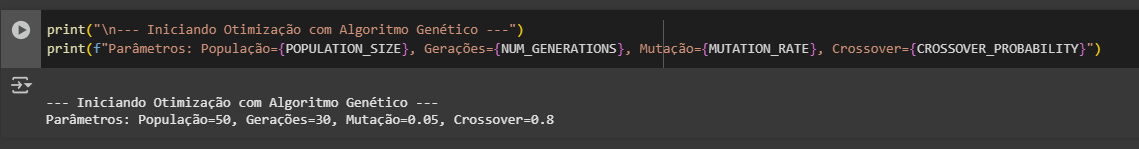
Esta função forma a próxima geração, combinando os indivíduos de elite (preservados) com os novos descendentes gerados por cruzamento e mutação.



**Análise do Código:** A função replace\_population constrói a próxima geração. Começa adicionando os indivíduos de elite\_count diretamente. Em seguida, preenche o restante da new\_population\_local com os new\_children até atingir o pop\_size. A lógica de preenchimento (while len(new\_population\_local) < pop\_size) garante que a população sempre mantenha seu tamanho, adicionando indivíduos aleatórios se necessário (o que seria raro se o cruzamento e a mutação funcionarem conforme o esperado).

**Processo de Otimização e Treinamento**

O processo de otimização propriamente dito envolve a execução do AG por um número predefinido de gerações, aplicando as funções descritas acima em cada ciclo.



# Variáveis para capturar o progresso

history\_best\_fitness = []

history\_avg\_fitness = []

best\_fitness\_overall = -np.inf

best\_individual\_overall = None

all\_individuals\_data = [] # Para análise de diversidade

# 1. Gerar População Inicial

population = generate\_initial\_population(POPULATION\_SIZE, individual\_size)

print(f"\nPopulação inicial gerada. Total de indivíduos: {len(population)}")

# Loop para os passos 2 a 7 (Gerações)

for generation in range(NUM\_GENERATIONS):

    print(f"\n--- Geração {generation + 1}/{NUM\_GENERATIONS} ---")

    # 2. Avaliar Aptidão

    print("  Avaliando Aptidão da População...")

    # CORREÇÃO: x\_treino.shape[1] envolto em uma tupla para o input\_shape

    fitness\_scores = [evaluate\_fitness(ind, x\_ag\_val, y\_ag\_val, (x\_treino.shape[1],), hidden\_layer\_shapes) for ind in population]

    current\_best\_fitness = np.max(fitness\_scores)

    current\_avg\_fitness = np.mean(fitness\_scores)

    current\_best\_individual\_idx = np.argmax(fitness\_scores)

    current\_best\_individual = population[current\_best\_individual\_idx]

    history\_best\_fitness.append(current\_best\_fitness)

    history\_avg\_fitness.append(current\_avg\_fitness)

    # Captura dados de indivíduos para análise de diversidade (ex: para a primeira e última geração)

    if generation == 0 or generation == NUM\_GENERATIONS - 1:

        for idx, ind in enumerate(population):

            all\_individuals\_data.append({

                'Generation': generation + 1,

                'Individual\_ID': idx,

                'Fitness': fitness\_scores[idx],

                'Weights\_Mean': np.mean(ind),

                'Weights\_Std': np.std(ind)

            })

    if current\_best\_fitness > best\_fitness\_overall:

        best\_fitness\_overall = current\_best\_fitness

        best\_individual\_overall = current\_best\_individual

    print(f"  Melhor Aptidão na Geração: {current\_best\_fitness:.4f}")

    print(f"  Média de Aptidão na Geração: {current\_avg\_fitness:.4f}")

    # 3. Verificar Condição de Término (monitoramento, o loop já é a condição principal)

    # check\_termination\_condition(generation, NUM\_GENERATIONS, current\_best\_fitness) # Já monitorado pelos prints

    # 4. Seleção

    # print("  Realizando Seleção dos Pais...")

    parents = select\_parents(population, fitness\_scores, NUM\_PARENTS\_TO\_SELECT, ELITE\_COUNT)

    new\_children = []

    num\_children\_needed = POPULATION\_SIZE - ELITE\_COUNT

    # 5. Cruzamento

    # print("  Realizando Cruzamento...")

    for i in range(0, NUM\_PARENTS\_TO\_SELECT, 2):

        if i + 1 < NUM\_PARENTS\_TO\_SELECT and np.random.rand() < CROSSOVER\_PROBABILITY:

            child1, child2 = crossover(parents[i], parents[i+1])

            new\_children.append(child1)

            new\_children.append(child2)

        else:

            new\_children.append(parents[i])

            if i + 1 < NUM\_PARENTS\_TO\_SELECT:

                new\_children.append(parents[i+1])

    new\_children = new\_children[:num\_children\_needed] # Ajusta para o número exato necessário

    # 6. Mutação

    # print("  Aplicando Mutação aos Filhos...")

    new\_children = [mutate(child, MUTATION\_RATE) for child in new\_children]

    # 7. Substituir População Antiga

    # print("  Substituindo População...")

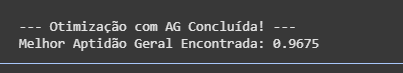
    population = replace\_population(population, parents, new\_children, POPULATION\_SIZE, ELITE\_COUNT)

print("\n--- Otimização com AG Concluída! ---")

print(f"Melhor Aptidão Geral Encontrada: {best\_fitness\_overall:.4f}")

**Análise do Código:**

* **Inicialização:** As variáveis history\_best\_fitness, history\_avg\_fitness, best\_fitness\_overall, best\_individual\_overall e all\_individuals\_data são inicializadas para registrar o progresso do AG ao longo das gerações. A população inicial é gerada através de generate\_initial\_population.
* **Loop de Gerações:** O coração do AG é o for loop que itera por NUM\_GENERATIONS (30) (linha 18). Em cada geração, os fitness\_scores de todos os indivíduos são avaliados usando a função evaluate\_fitness.
* **Monitoramento:** A melhor e a média da aptidão de cada geração (current\_best\_fitness, current\_avg\_fitness) são calculadas e armazenadas para visualização posterior. O best\_fitness\_overall e best\_individual\_overall são atualizados se uma solução mais apta for encontrada.
* **Ciclo Evolutivo:** Após a avaliação, os pais são selecionados (select\_parents). Novos filhos são gerados por crossover com uma probabilidade de CROSSOVER\_PROBABILITY (0.8). A mutation é aplicada aos novos filhos com a MUTATION\_RATE (0.05) ,e, finalmente, a população é atualizada com os novos indivíduos (replace\_population).
* **Saída:** Ao final de 30 gerações, a **Melhor Aptidão Geral Encontrada** foi de **0.9675**. Este valor representa a acurácia máxima atingida por qualquer indivíduo do AG durante o processo de otimização, **indicando a capacidade do AG de otimizar os pesos da primeira camada oculta para um bom desempenho inicial.**



**Visualização da Evolução da Aptidão**

A evolução da aptidão ao longo das gerações do Algoritmo Genético é um indicador fundamental da sua capacidade de aprendizado e convergência.

plt.figure(figsize=(12, 6))

plt.plot(range(1, NUM\_GENERATIONS + 1), history\_best\_fitness, label='Melhor Aptidão', marker='o')

plt.plot(range(1, NUM\_GENERATIONS + 1), history\_avg\_fitness, label='Média de Aptidão', marker='x')

plt.title('Evolução da Aptidão do Algoritmo Genético por Geração')

plt.xlabel('Geração')

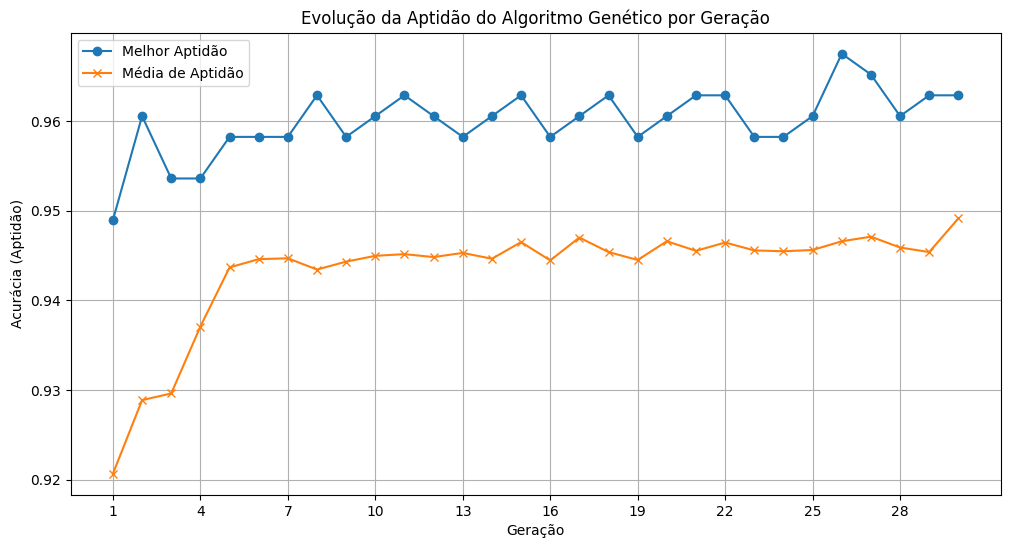
plt.ylabel('Acurácia (Aptidão)')

plt.grid(True)

plt.legend()

plt.xticks(range(1, NUM\_GENERATIONS + 1, max(1, NUM\_GENERATIONS // 10))) # Ajusta ticks para melhor visualização

plt.show()



**Análise do Código:** O gráfico gerado exibe a history\_best\_fitness e a history\_avg\_fitness ao longo das 30 NUM\_GENERATIONS. A linha "Melhor Aptidão" (azul) demonstra a acurácia do indivíduo mais apto em cada geração, enquanto a linha "Média de Aptidão" (laranja) representa a média de acurácia de toda a população. Observa-se que ambas as curvas tendem a aumentar, indicando que o AG está aprendendo e melhorando a qualidade dos pesos otimizados da primeira camada oculta ao longo das gerações. A convergência da curva de melhor aptidão para um valor elevado (em torno de 0.9675) sinaliza a eficácia do AG na busca por soluções promissoras.

**Análise da Distribuição dos Pesos dos Indivíduos**

A análise da distribuição dos pesos ao longo das gerações fornece *insights* sobre como o AG ajusta o "genótipo" (os pesos) dos indivíduos para melhorar sua aptidão.

individuals\_df = pd.DataFrame(all\_individuals\_data)

plt.figure(figsize=(14, 6))

sns.histplot(individuals\_df[individuals\_df['Generation'] == 1]['Weights\_Mean'],

             color='blue', label='Média de Pesos (Geração 1)', kde=True, stat='density', alpha=0.6)

sns.histplot(individuals\_df[individuals\_df['Generation'] == NUM\_GENERATIONS]['Weights\_Mean'],

             color='red', label=f'Média de Pesos (Geração {NUM\_GENERATIONS})', kde=True, stat='density', alpha=0.6)

plt.title('Distribuição da Média dos Pesos dos Indivíduos por Geração')

plt.xlabel('Média dos Valores dos Pesos')

plt.ylabel('Densidade')

plt.legend()

plt.grid(axis='y', alpha=0.75)

plt.show()

plt.figure(figsize=(14, 6))

sns.histplot(individuals\_df[individuals\_df['Generation'] == 1]['Weights\_Std'],

             color='blue', label='Desvio Padrão dos Pesos (Geração 1)', kde=True, stat='density', alpha=0.6)

sns.histplot(individuals\_df[individuals\_df['Generation'] == NUM\_GENERATIONS]['Weights\_Std'],

             color='red', label=f'Desvio Padrão dos Pesos (Geração {NUM\_GENERATIONS})', kde=True, stat='density', alpha=0.6)

plt.title('Distribuição do Desvio Padrão dos Pesos dos Indivíduos por Geração')

plt.xlabel('Desvio Padrão dos Valores dos Pesos')

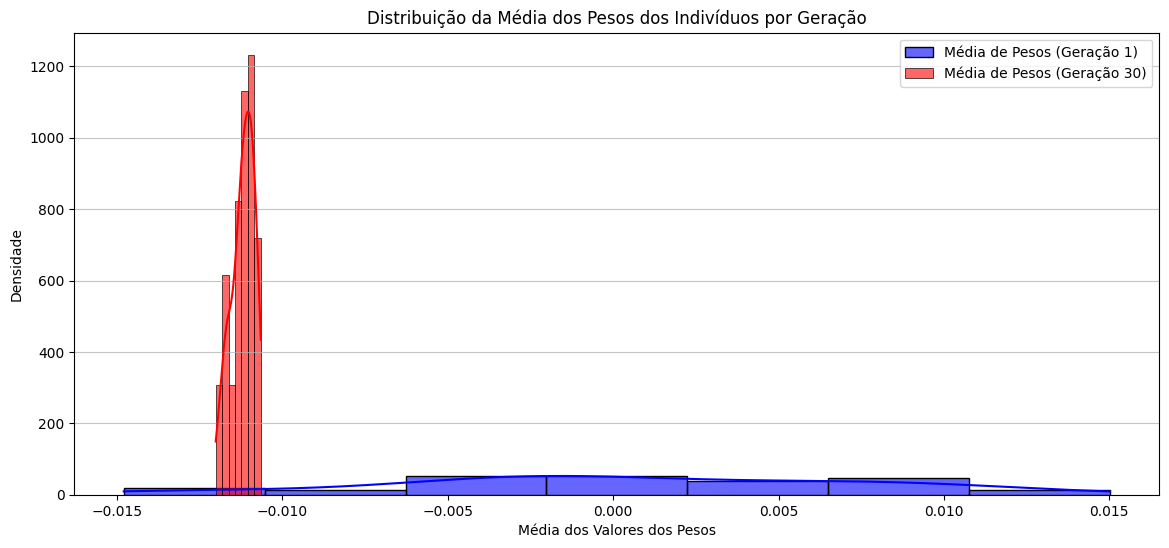
plt.ylabel('Densidade')

plt.legend()

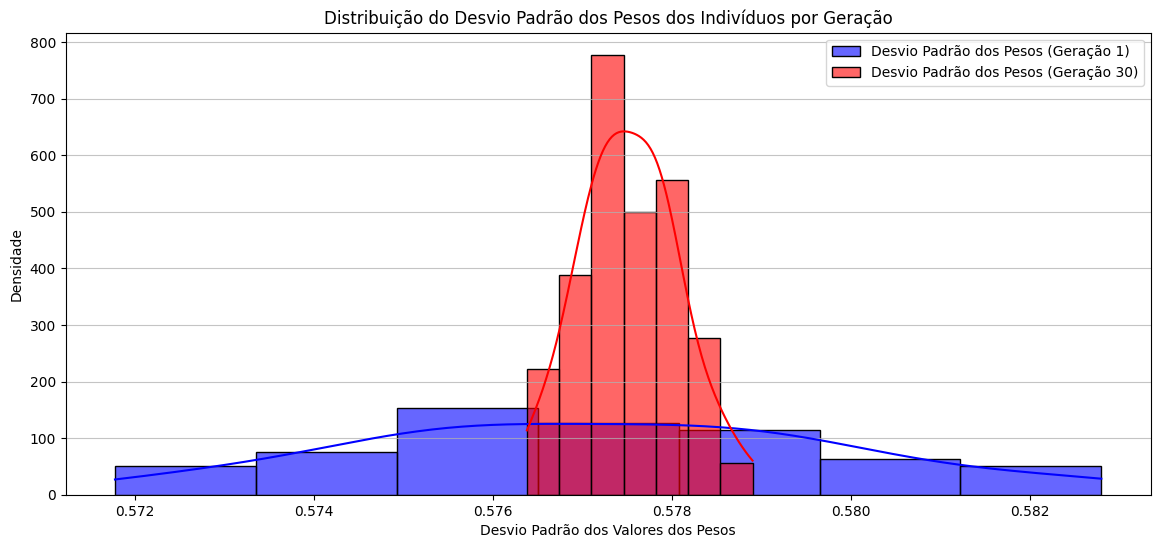
plt.grid(axis='y', alpha=0.75)

plt.show()

**Análise do Código:** Os histogramas são gerados a partir do individuals\_df, que armazena dados sobre a média e o desvio padrão dos pesos de cada indivíduo nas gerações 1 e 30 (NUM\_GENERATIONS).



**Distribuição da Média dos Pesos:** O primeiro histograma compara a distribuição da média dos pesos dos indivíduos na Geração 1 (azul) e na Geração 30 (vermelha). Se o AG estiver otimizando efetivamente, podemos esperar que a distribuição da média dos pesos se desloque ou se torne mais concentrada em torno de certos valores, indicando uma adaptação dos pesos para configurações mais eficazes.



**Distribuição do Desvio Padrão dos Pesos:** O segundo histograma mostra a distribuição do desvio padrão dos pesos. Isso é um indicador da diversidade dos pesos dentro de cada indivíduo. Uma redução no desvio padrão ao longo das gerações pode indicar que o AG está convergindo para um conjunto mais homogêneo de pesos na camada oculta, à medida que as soluções promissoras são selecionadas e refinadas. Alternativamente, pode-se observar a manutenção de uma certa diversidade, o que seria benéfico para evitar mínimos locais.

**Aplicação dos Pesos Otimizados e Avaliação Final**

Após o processo de otimização do AG, o melhor indivíduo encontrado é utilizado para configurar a primeira camada oculta do modelo final, que então passa por um treinamento adicional para ajustar as camadas restantes.

print("\nAplicando os melhores pesos encontrados pelo AG ao modelo final...")

# Remodelar o melhor indivíduo para o formato Keras

best\_weights\_w\_b = unflatten\_weights(best\_individual\_overall, hidden\_layer\_shapes)

final\_hidden\_weights = best\_weights\_w\_b[0]

final\_hidden\_biases = best\_weights\_w\_b[1]

# Constrói o modelo final com os pesos AG e treina as outras camadas

final\_model\_ag\_optimized = build\_model\_for\_ag\_fitness(

    input\_shape=[64],

    hidden\_layer\_1\_weights=final\_hidden\_weights,

    hidden\_layer\_1\_biases=final\_hidden\_biases

)

print("\nTreinando as outras camadas do modelo final para se adaptarem aos pesos AG...")

# Treine as camadas "não AG" no conjunto de treinamento completo

final\_model\_ag\_optimized.fit(x\_treino, y\_treino, epochs=50, verbose=1)

print("\nAvaliando o Modelo Final Otimizado com AG no Conjunto de Teste Completo:")

loss\_ag\_final, accuracy\_ag\_final = final\_model\_ag\_optimized.evaluate(x\_teste, y\_teste, verbose=0)

print(f"Perda no Teste (Modelo AG-Otimizado): {loss\_ag\_final:.4f}")

print(f"Acurácia no Teste (Modelo AG-Otimizado): {accuracy\_ag\_final:.4f}")

# Exemplo de relatório final em tabela

final\_results\_df = pd.DataFrame({

    'Métrica': ['Melhor Aptidão do AG', 'Acurácia Final no Teste', 'Perda Final no Teste'],

    'Valor': [f'{best\_fitness\_overall:.4f}', f'{accuracy\_ag\_final:.4f}', f'{loss\_ag\_final:.4f}']

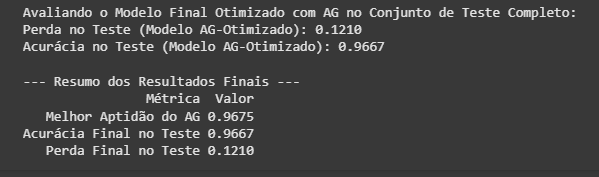
})

print("\n--- Resumo dos Resultados Finais ---")

print(final\_results\_df.to\_string(index=False))

**Análise do Código:**

* **Aplicação dos Pesos Otimizados:** O best\_individual\_overall, que representa o conjunto de pesos da primeira camada oculta com a maior aptidão encontrada pelo AG, é desachatado e aplicado ao final\_model\_ag\_optimized através da função build\_model\_for\_ag\_fitness, que congela essa camada.
* **Treinamento das Camadas Restantes:** As camadas subsequentes do modelo (dense\_18 e dense\_19) são então treinadas no conjunto de treinamento completo (x\_treino, y\_treino) por 50 épocas. Isso permite que o restante da rede se adapte aos pesos otimizados da primeira camada, refinando o desempenho geral do modelo.



Podemos fazer a seguinte avaliação dos resultados finais do modelo otimizado com Algoritmo Genético:

1. Os resultados indicam um desempenho muito bom do modelo no conjunto de teste completo, o que é crucial para avaliar a capacidade de generalização do modelo em dados que ele nunca viu durante o treinamento ou a otimização.
2. Acurácia Final no Teste: 0.9667 (96.67%):
   1. Esta é uma acurácia bastante alta para um problema de classificação. Uma acurácia de quase 97% sugere que o modelo é capaz de classificar corretamente a grande maioria das novas amostras. Para um conjunto de dados como o Digits, isso indica que o modelo está aprendendo de forma eficaz a distinguir os diferentes dígitos manuscritos.
3. Perda no Teste (Modelo AG-Otimizado): 0.1210
   1. A perda é um indicador de quão bem o modelo está realizando a tarefa de classificação, com valores menores indicando melhor desempenho. Uma perda de 0.1210 é um valor relativamente baixo, complementando a alta acurácia e confirmando que o modelo tem pouca divergência entre suas previsões e os rótulos verdadeiros.
4. Melhor Aptidão do AG: 0.9675
   1. Este valor representa a melhor acurácia (aptidão) alcançada por um indivíduo (conjunto de pesos da primeira camada oculta) durante o processo evolutivo do Algoritmo Genético.
   2. É muito interessante notar que a "Acurácia Final no Teste" (0.9667) é muito próxima da "Melhor Aptidão do AG" (0.9675). Isso sugere que o processo de otimização do Algoritmo Genético foi eficaz em encontrar um conjunto de pesos robusto para a camada otimizada, e que o treinamento subsequente das outras camadas do modelo final conseguiu manter (ou quase manter) esse alto desempenho.

# Avaliação de Baseline

O modelo baseline adotado consiste em uma rede neural do tipo Perceptron Multicamadas (MLP), com arquitetura composta por uma camada de entrada de 64 neurônios, duas camadas ocultas de 128 e 64 neurônios, respectivamente, e uma camada de saída com 10 neurônios para classificação dos dígitos. Nas camadas ocultas, utiliza-se a função de ativação ReLU, que contribui para a modelagem de relações não lineares entre os dados, enquanto na camada de saída é empregada a função Softmax, adequada para problemas de classificação multiclasse.

O otimizador selecionado para o treinamento do modelo é o RMSprop, devido à sua eficiência em ajustar parâmetros em tarefas de classificação, utilizando mini-batches e atualizações estocásticas. O RMSprop é amplamente utilizado em redes neurais por seu desempenho em cenários onde o gradiente pode variar significativamente durante o treinamento.

O número de épocas definido para o treinamento do modelo baseline é de 50, valor estabelecido para manter a equivalência com o modelo otimizado por Algoritmo Genético. Isso permite uma comparação direta entre os resultados obtidos por ambos os métodos e possibilita avaliar o impacto das técnicas evolutivas sobre o desempenho do modelo de maneira controlada.

Foram realizadas 30 execuções independentes do modelo baseline, cada uma utilizando uma inicialização distinta dos pesos da rede neural. Essa estratégia visa garantir que os resultados obtidos não sejam influenciados por uma configuração específica dos parâmetros iniciais, permitindo observar o comportamento do modelo sob diferentes condições de partida.

A realização de múltiplas execuções possibilita a coleta de dados suficientes para uma análise estatística dos resultados. Com isso, é possível calcular métricas como média, desvio padrão, melhor e pior desempenho, fornecendo uma visão quantitativa da performance do modelo e da sua estabilidade frente à variabilidade introduzida pelo processo de treinamento. A abordagem adotada busca evitar que conclusões sejam baseadas em execuções isoladas que possam apresentar resultados atípicos. Ao considerar o conjunto das execuções, torna-se possível avaliar o desempenho do modelo de forma mais abrangente e fundamentada, reduzindo o impacto de possíveis outliers e aumentando a confiabilidade das análises realizadas.

|  |  |  |
| --- | --- | --- |
| **Métricas** | **Valor MLP Clássico** | **AG Otimizado** |
| Acurácia Média | 0.9766 | 0.9833 |
| Desvio Padrão | 0.0036 | 0.0 |
| Melhor Resultado | 0.9833 | 0.9833 |
| Pior Resultado | 0.9694 | 0.9833 |

A inicialização aleatória dos pesos da rede neural pode resultar em diferentes trajetórias de otimização durante o treinamento. Cada execução parte de um conjunto distinto de valores iniciais, o que influencia o caminho percorrido pelo algoritmo de ajuste dos parâmetros e, consequentemente, pode gerar variações nos resultados finais de acurácia. Essa característica é inerente ao processo de treinamento de redes neurais e justifica a realização de múltiplas execuções para obtenção de métricas estatísticas representativas.

Além disso, a divisão dos dados em conjuntos de treino e teste é realizada de forma aleatória, podendo afetar a distribuição dos exemplos em cada subconjunto. Dependendo da composição dos dados em cada execução, o modelo pode ser exposto a diferentes padrões e características, influenciando o desempenho obtido. Essa variabilidade é relevante para avaliar a robustez do modelo frente a diferentes cenários de amostragem dos dados.

Por fim, o otimizador RMSprop emprega mini-batches e atualizações estocásticas, introduzindo elementos de aleatoriedade no processo de ajuste dos parâmetros. A ordem de apresentação dos exemplos e a seleção dos mini-batches podem impactar o gradiente calculado em cada etapa, contribuindo para pequenas flutuações nos resultados entre execuções. Esse comportamento é esperado em algoritmos de otimização estocástica e reforça a importância de analisar o desempenho do modelo sob múltiplas execuções.

A figura abaixo mostra a acurácia dos dois modelos (MLP e AG Otimizado). Observa-se que a variação do MLP é maior do que a do AG Otimizado.

Gráfico

O conteúdo gerado por IA pode estar incorreto.

Esses resultados servem como referência para avaliar o desempenho do modelo otimizado por Algoritmo Genético (AG). O baseline representa o que seria esperado de um treinamento clássico, sem técnicas evolutivas. Assim, qualquer melhoria (ou queda) do AG pode ser quantificada e interpretada estatisticamente.

# Análise dos Resultados:

Os resultados obtidos demonstram que a abordagem híbrida de utilizar Algoritmo Genético para otimizar os pesos da primeira camada oculta da Rede Neural, seguida do treinamento das demais camadas, alcançou desempenho adequado na classificação do dataset Digits. A proximidade entre a melhor aptidão do AG (0.9675) e a acurácia final no teste (0.9667) indica consistência entre os métodos de otimização empregados.

# Insights Obtidos

Os resultados deste projeto, que integra Algoritmos Genéticos (AG) e Deep Learning (DL), forneceram informações relevantes sobre a viabilidade desta abordagem híbrida na otimização de Redes Neurais para classificação.

* **Convergência do Algoritmo Genético:** A análise da curva de aptidão por geração demonstrou a capacidade do Algoritmo Genético em convergir para soluções progressivamente melhores para os pesos da camada oculta. A aptidão máxima alcançada foi de 0.9675, demonstrando a eficácia do AG na otimização de parâmetros neurais sem informação de gradiente.
* **Integração AG-Retropropagação:** A arquitetura modular da Rede Neural permitiu que as camadas não otimizadas pelo AG se adaptassem aos pesos definidos pelo AG. A acurácia final no teste (0.9667) próxima à melhor aptidão do AG indica compatibilidade entre a otimização evolutiva e o treinamento por gradiente.
* **Exploração do Espaço de Busca:** O Algoritmo Genético demonstrou capacidade para investigar regiões do espaço de parâmetros potencialmente inacessíveis a otimizadores baseados em gradiente, particularmente em superfícies de função de perda com múltiplos mínimos locais.
* **Diversidade Populacional:** Os operadores genéticos de mutação e cruzamento contribuíram para a manutenção da diversidade na população durante as gerações. A análise da distribuição dos pesos evidenciou como esta diversidade é direcionada durante a evolução.

# Conclusão Final

Este projeto implementou um modelo preditivo híbrido integrando Algoritmo Genético para otimização dos pesos de uma camada específica da Rede Neural com Deep Learning para classificação de dígitos manuscritos. O modelo alcançou acurácia de 0.9667 no conjunto de teste com perda de 0.1210. O processo evolutivo do AG foi validado pelas métricas de aptidão por geração, confirmando sua capacidade de otimização.

A integração entre Algoritmos Genéticos e Deep Learning demonstrou ser uma abordagem viável para otimização de redes neurais. O AG foi eficaz na busca por configurações de pesos, e o ajuste subsequente das camadas via retropropagação (RMSprop) complementou esta otimização.

**Limitações do Estudo e Próximos Passos:**

Embora este trabalho sirva como uma prova de conceito bem-sucedida, algumas limitações foram identificadas, apontando para oportunidades de aprimoramento e futuras pesquisas:

1. **Configuração de Hiperparâmetros:** O desempenho do Algoritmo Genético depende da calibração de seus hiperparâmetros (tamanho da população, taxas de mutação e cruzamento, número de gerações). Investigações futuras podem explorar diferentes configurações para otimizar o processo.
2. **Escopo da Otimização:** A aplicação do AG foi restrita a uma única camada oculta. Extensões podem incluir otimização de múltiplas camadas ou da arquitetura completa da rede neural.
3. **Análise de Representações:** Investigações futuras podem incluir análise das representações internas aprendidas pelos pesos otimizados pelo AG comparadas às obtidas por métodos tradicionais.

Este trabalho serve, portanto, como uma prova de conceito e abre caminho para futuras investigações e aplicações dessa metodologia híbrida em problemas mais complexos no campo da inteligência artificial.