Reconhecimento de Formas Geométricas com Rede Neural Feedforward Simples

Heitor Barcellos

20/06/2025

1. Introdução

1.1 Motivação

A capacidade de aprendizagem das máquinas é um dos fenômenos mais fascinantes e revolucionários da tecnologia moderna. Desde a popularização dos *chatbots* por volta de 2020, sempre me despertou curiosidade ver como programas são capazes de compreender dados, encontrar padrões e responder de forma semelhante à humana.

Este relatório vai além de explicar um código: meu objetivo é tornar o conceito de redes neurais acessível para quem nunca teve contato com o tema, mostrando que é possível entender os fundamentos básicos. Para isso, cada módulo do programa será detalhado de forma didática, acompanhando algumas explicações visuais e exemplos práticos.

A base teórica do trabalho à qual vamos recorrer para explicações, é o capítulo 7 do livro *Problemas Clássicos de Ciência da Computação com Python*, de David Kopec — uma recomendação literária do professor. A ideia é combinar teoria, prática e uma linguagem fácil, aproximando o leitor leigo ao universo do *machine learning*.

1.2 Sobre a Organização do Projeto

O projeto desenvolvido em Python é estruturado de forma modular, seguindo a ideia do projeto interior, de que um código organizado facilita o entendimento ou futuras manutenções, tanto para quem está desenvolvendo quanto para quem deseja compreender o código.

Vamos começar a analisar a estrutura do sistema e o papel de cada módulo dentro do fluxo geral. O projeto está dividido em três pastas, sendo as duas principais: **`src`** (onde a lógica de processamento da rede neural e do treinamento ocorre) e **`data`** (onde os dados necessários para treinamento são armazenados). Abaixo, vamos detalhar a função de cada pasta e arquivo, explicando como eles se relacionam entre si para formar o sistema como um todo.

1.2.1 Pasta `src/`

Dentro da pasta, temos os módulos principais que compõem a lógica da rede neural e o processo de treinamento.

O arquivo `network.py` define a classe RedeNeural, que organiza a rede e controla o fluxo de dados entre as camadas. Também é responsável por orquestrar o treinamento e a validação da rede neural. A classe Camada, presente em `layer.py` , gerencia as camadas da rede.

Já o arquivo `neuron.py` contém a classe `Neuronio`, que simula o funcionamento de um neurônio, calculando a soma ponderada das entradas, aplicando a função de ativação e gerando a saída que será passada para a camada seguinte.

O arquivo `util.py`, oferece funções auxiliares, que também são fundamentais para o funcionamento da rede, como o cálculo de produto escalar, a aplicação da função de ativação e sua derivada, além de uma função de normalização de dados que prepara as entradas — os termos citados, que podem parecer novos para o leitor de início, serão explicados com mais detalhes nos próximos tópicos.

Em alguns dos módulos, aplicamos a **Programação Orientada a Objetos** (POO), o que significa que usamos classes e métodos para organizar a estrutura do código. Portanto, ao explicar os módulos a seguir, alguns conceitos específicos de POO poderão ser usados para descrever a estrutura da rede neural.

Os arquivos `dataset_imagem.py` e `gerar_imagens.py` são responsáveis pela criação e por processar os dados de treinamento. `dataset_imagem.py` carrega as imagens de formas geométricas armazenadas na pasta *data* e as converte em vetores numéricos. O `gerar_imagens.py`, por sua vez, cria automaticamente essas imagens e as salva na pasta *data*.

Por último temos a cereja do bolo, o arquivo `limitrofes.py`, responsável pela criação e teste de imagens híbridas (casos limítrofes). O módulo testa a capacidade da rede em classificar imagens que combinam diferentes formas geométricas, desafiando a rede a generalizar além das formas puras. Contém funções para gerar essas imagens e processá-las durante os testes.

Apesar dos últimos três arquivos estarem nesta pasta, eles estão relacionados à nossa pasta *data*, que faremos uma breve explicação a seguir.

O código completo do projeto está disponível em: https://github.com/pbheitor/redeneural-simples

1.2.2 Pasta `data/`

A pasta data pode ser vista como auxiliar, pois ela armazena os dados de entrada necessários para o treinamento/validação do modelo. Ela não possui lógica ou controle de fluxo do sistema, mas fornece imagens, que são usadas pela rede neural para aprender e prever resultados.

1.2.3 Pasta `venv/`

A pasta é um ambiente virtual criado para instalar a biblioteca **Pillow** (sucessora da PIL), garantindo que a biblioteca funcione de forma isolada, sem a necessidade de instalação global diretamente do sistema operacional.

2. Entendendo uma Rede Neural *Feedforward* Simples

Antes de começarmos a explicar como os módulos funcionam e se relacionam, é interessante que você entenda alguns conceitos, como qual é o modelo desse tipo de rede neural. No projeto, vamos trabalhar com uma rede neural *feedforward* simples, um tipo básico de rede usada para classificação.

A ideia é simples: ela pega uma entrada (como uma imagem ou qualquer outro tipo de dado), processa essas informações em camadas, e então dá uma resposta final. O nome *feedforward* significa que os dados fluem apenas em uma direção: da entrada para a saída. Na **Figura 1**, pode-se observar um diagrama simples de como a rede funciona, com o fluxo de dados passando pelas camadas até gerar a saída final.

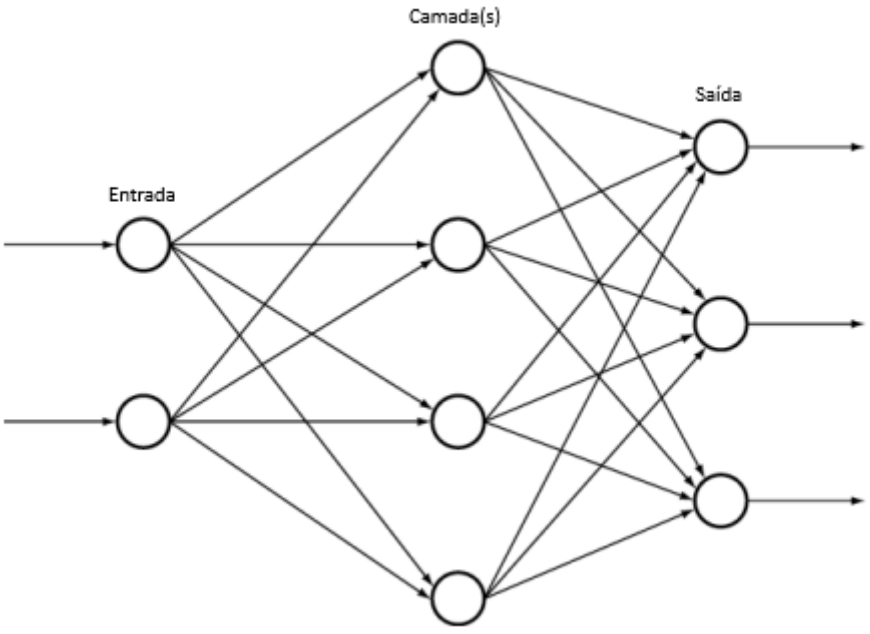

Figura 1 – A imagem ilustra de forma superficial uma rede neural *feedforward* simples, onde os dados fluem da entrada para as camadas e por fim, geram uma resposta na saída.

2.1 O que é um neurônio em uma Rede Neural?

Em uma rede neural, o neurônio é a unidade básica de processamento. Ele funciona de forma análoga aos neurônios no cérebro humano, recebendo entradas, processando essas informações e gerando uma saída. Cada neurônio da rede realiza uma operação matemática simples: ele recebe múltiplas entradas. Cada entrada é multiplicada por um peso correspondente, que determina a importância daquela entrada no cálculo.

Essas entradas são somadas, essa soma chamamos de **soma ponderada**, e o resultado é passado por uma **função de ativação**, (e.g. a função sigmoide) que transforma o valor de forma a permitir que a rede consiga entender padrões mais complexos. Sem essa função, a rede seria capaz de aprender apenas relações simples e lineares. A função de ativação, no entanto, ajuda a rede a aprender padrões mais complexos, como reconhecer imagens, sons ou outros tipos de dados.

O valor final gerado pela função de ativação é a **saída do neurônio**, que pode ser utilizada como entrada para os neurônios das camadas seguintes ou, na camada de saída, representar a previsão final do modelo.

O ajuste dos pesos ocorre durante o treinamento da rede, permitindo que ela aprenda e melhore sua capacidade de prever ou classificar dados de forma cada vez mais precisa. Toda explicação acima pode ser compreendida melhor e de forma ilustrada na **Figura 2**, abaixo.

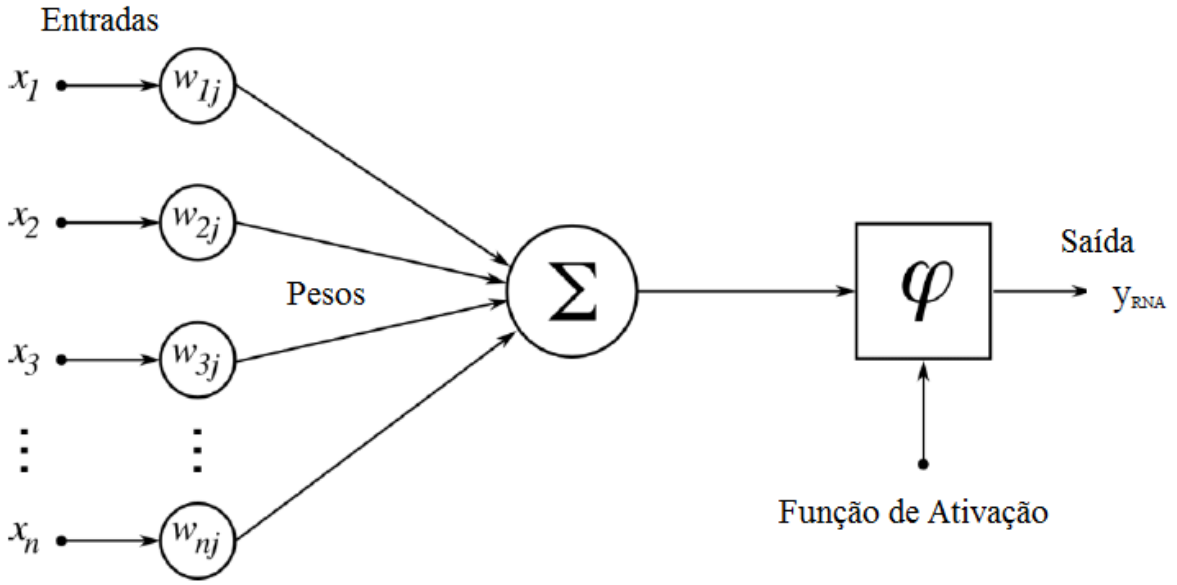

Figura 2 - Representação de um neurônio artificial e seus componentes. É didático perceber que, ao receber as entradas (x_1, x_2, x_3 [...] $x\square$), o neurônio as multiplica pelos respectivos pesos. Ou seja, w_1j representa o peso associado à entrada x_1, onde o peso ajusta a importância de cada entrada no cálculo do neurônio.

2.2 Classe RedeNeural | `network.py`

Agora que os conceitos de redes neurais e de neurônios artificiais foram introduzidos, é importante destacar que a classe `RedeNeural` é o núcleo do nosso algoritmo. Ela não apenas controla o fluxo de dados, mas também orquestra o treinamento da rede, garantindo que o modelo aprenda com os dados fornecidos. Vamos explorar essa classe, abordando seus principais métodos e como ela interage com as camadas da rede.

2.2.1 Inicialização da Rede | `__init__()`

Começamos pelo construtor da classe, responsável por inicializar a estrutura da rede neural. Durante a inicialização, a classe recebe informações como número de neurônios por camada e taxa de aprendizado. Além disso, a rede é configurada por camadas, cada camada é criada de acordo com o número de neurônios especificado para cada uma. A primeira camada é criada com entradas e a última com saída, com camadas ocultas entre elas, como vimos anteriormente na **Figura 1**, com a observação que, "Entrada" estamos nos referindo à camada de entrada, enquanto "Camada(s)", são as camadas ocultas, e "Saída", a camada de saída.

```python
class RedeNeural:

    def __init__(self, estruturaCamadas, taxaAprendizagem, funcaoAtivacao=sigmoide, derivadaAtivacao=derivadaSigmoide):
        if len(estruturaCamadas) < 3:
            raise ValueError("A rede precisa de ao menos 3 camadas (entrada, oculta, saída)")
        self.camadas = []
        camadaEntrada = Camada(None, estruturaCamadas[0], taxaAprendizagem, funcaoAtivacao, derivadaAtivacao)
        self.camadas.append(camadaEntrada)
        for i, numNeuronios in enumerate(estruturaCamadas[1:]):
            novaCamada = Camada(self.camadas[i], numNeuronios, taxaAprendizagem, funcaoAtivacao, derivadaAtivacao)
            self.camadas.append(novaCamada)


    def calcularSaidas(self, entrada):
        return reduce(lambda entradas, camada: camada.calcularSaidas(entradas), self.camadas, entrada)


    def retropropagar(self, esperado):
        ultima = len(self.camadas) - 1
        self.camadas[ultima].calcularDeltasSaida(esperado)
        for l in range(ultima - 1, 0, -1):
            self.camadas[l].calcularDeltasOculta(self.camadas[l + 1])


    def atualizarPesos(self):
        for camada in self.camadas[1:]:
            for neuronio in camada.neuronios:
                for w in range(len(neuronio.pesos)):
                    neuronio.pesos[w] += neuronio.taxaAprendizagem * camada.camadaAnterior.saidasCache[w] * neuronio.delta
```

Figura 3 – Na imagem, podemos observar quatro métodos da classe `RedeNeural`.

A função recebe quatro parâmetros. O parâmetro `estruturaCamadas`, é uma lista que define a quantidade de neurônios em cada camada (entrada,

camadas ocultas e saída), enquanto `taxaAprendizagem` controla o quanto os pesos serão ajustados a cada iteração. Já `funcaoAtivacao`, é nossa função de ativação usada nos neurônios, `derivadaAtivacao` é a derivada da função de ativação, que é usada na retropropagação para ajustar os pesos.

Agora, vamos conseguir entender o que de fato acontece no código do método. Primeiro, fazemos uma validação da estrutura da rede, que deve ter no mínimo 3 camadas. Em seguida, a primeira camada é criada, com base nas entradas, como não tem uma camada anterior, ela recebe *None*. Por fim, para cada camada oculta e de saída, a classe `Camada` é conectada à camada anterior. Cada camada é adicionada à lista `self.camadas`, que armazena todas as camadas da rede.

A complexidade dessa função é O(n), onde *n* é o número de neurônios na camada de entrada, pois cada camada é inicializada em relação ao número de neurônios da camada anterior.

2.2.2 Cálculo das Saídas da Rede | `calcularSaidas()`

Logo após o método construtor, temos o método que calcula as saídas da rede, você pode observar na **Figura 3**. O método só recebe um parâmetro, que são `entradas`, dados que serão processados pela rede neural (no nosso caso, vetor de pixels de uma imagem).

Então, o método retorna uma função chamada `reduce()`, uma função da biblioteca `functools`, que é utilizada para aplicar uma mesma operação

repetidamente em uma sequência, acumulando um resultado final. Normalmente *reduce* recebe três parâmetros, que são: função, a sequência e o valor inicial do acumulador. No nosso projeto, poderíamos entender da seguinte forma:

- **Função:** seria nossa função lambda, que é definida no momento de uso, neste método, a função lambda recebe dois parâmetros, entradas (valor acumulado) e camada, que é cada item da lista `self.camadas`.
- **Sequência**: é a lista `self.camadas`, que contém todas as camadas da rede neural.
- **Valor inicial**: corresponde às entradas iniciais fornecidas ao método. A cada camada percorrida, esse valor é atualizado para a saída da camada atual, e o resultado passa a ser a entrada para a próxima camada.

A complexidade desta função é O(m * n), onde *m* é o número de camadas e *n* é o número de neurônios, pois o cálculo das saídas envolve a iteração por todas as camadas e neurônios da rede.

2.2.3 Retropropagação do Erro | `retropropagar()`

Para quem não está familiarizado com redes neurais, talvez não saiba o que é a retropropagação. A retropropagação é o processo fundamental pelo qual a rede neural ajusta os pesos durante o treinamento. Ela ocorre após a

rede calcular a saída para uma entrada específica. Ela calcula o erro (que seria a diferença entre a saída real e a esperada) e propaga esse erro de volta pela rede, camada por camada. Esse erro é então usado para ajustar os pesos, fazendo que a rede se torne mais precisa a cada iteração. Agora com o conceito já familiarizado, vamos detalhar o método.

O método recebe como parâmetro apenas a saída esperada. Ao analisar o comportamento na **Figura 3**, vemos que a variável `ultima` determina o índice da última camada da rede. Em seguida, um método é chamado na última camada (`calcularDeltasSaida`), que calcula a diferença, então fazemos um loop que percorre as camadas de trás para frente, isso é importante porque a retropropagação começa da última camada e propaga o erro de volta através da rede. Por fim, para cada camada oculta, delta é calculado com base no erro da camada seguinte, este valor é utilizado para ajustar os pesos de cada neurônio na camada oculta.

A complexidade dessa função é O(n * m), onde *n* é o número de camadas e *m* é o número de neurônios por camada, já que é necessário propagar o erro de volta através de todas as camadas.

2.2.4 Atualização dos Pesos na Rede | `atualizarPesos()`

O método não recebe parâmetros, pois opera utilizando os dados já armazenados na classe. O objetivo do método é ajustar os pesos dos neurônios após a retropropagação, com base nos erros. Vale destacar que esse processo

de atualização é uma aplicação do algoritmo de **gradiente descendente**. Para uma explicação mais detalhada, consulte a página 199 do Capítulo 7 do livro utilizado como base teórica, onde a abordagem é melhor.

Começamos a iterar sobre as camadas da rede. É possível observar na **Figura 3** que a iteração começa pela segunda camada, porque a camada de entrada não precisa de ajuste de pesos, só recebe dados. Em seguida, iteramos sobre os neurônios de cada camada e, para cada um deles, percorremos seus pesos para realizar a atualização.

Complexidade dessa função é O(n * p), onde *n* é o número de camadas e *p* o número de neurônios por camada, devido à necessidade de atualizar os pesos de cada neurônio com base nos deltas calculados.

2.2.5 Treinamento da Rede | `treinar()`

O método é responsável por gerenciar o processo de aprendizado da rede neural, onde ela vai ajustando seus pesos com base nas entradas fornecidas e nas saídas esperadas.

```python
def treinar(self, entradas, esperados):
    for i, xs in enumerate(entradas):
        ys = esperados[i]
        self.calcularSaidas(xs)
        self.retropropagar(ys)
        self.atualizarPesos()


def validar(self, entradas, esperados, interpretarSaida):
    corretos = 0
    for entrada, esperado in zip(entradas, esperados):
        resultado = interpretarSaida(self.calcularSaidas(entrada))
        if resultado == esperado:
            corretos += 1
    percentual = corretos / len(entradas)
    return corretos, len(entradas), percentual
```

Figura 4 – Os últimos dois métodos da classe RedeNeural.

Ele recebe dois parâmetros, que são `entradas` e `esperados`, como é visto na **Figura 4**. O método começa iterando sobre as entradas fornecidas, com a variável `xs` representando cada entrada dos dados. Ao mesmo tempo, pega a saída esperada (`ys`), que corresponde à entrada atual.

Então é chamado o método de cálculo das saídas, para calcular as saídas para a entrada `xs` fornecida. A rede processa a entrada e passa os dados pelas camadas, realizando o cálculo de cada neurônio e gerando a saída prevista.

Após calcular a saída, é chamado o método de retropropagação, onde `ys` é a saída esperada. O método calcula o erro da rede, que é o delta. O erro é propagado de volta através da rede, ajustando os pesos de cada neurônio. Os

pesos são atualizados ao chamar o método `atualizarPesos()` para ajustar os pesos com base no erro propagado.

A complexidade dessa função é O(k * n * m), onde *k* é o número de épocas de treinamento, *n* o número de entradas, e *m* o número de neurônios nas camadas, pois cada entrada é processada em todas as camadas durante o treinamento.

2.2.6 Validação da Rede | `validar()`

A validação é o processo de avaliar a performance da rede neural após o treinamento. Ela permite verificar como a rede se comporta com um conjunto de dados que ela não viu durante o treinamento.

Recebe três parâmetros, dois já foram citados anteriormente, que são `entradas` e `esperados`, também temos `interpretarSaida`, que é uma função usada para interpretar a saída da rede e convertê-la em um valor de decisão, veja com mais detalhes na **Figura 4**.

Iniciamos com um contador chamado `corretos`, para contar quantas previsões a rede acertou. Em seguida, temos uma iteração em duas variáveis, percorrendo todas entradas e suas saídas esperadas. Para cada entrada, o método que calcula saídas é chamado, propagando a entrada pela rede, passando os dados pelas camadas até obter a previsão final.

A função `interpretarSaida()` converte a saída da rede (que geralmente é um valor probabilístico) em uma decisão. Então a saída interpretada `resultado` é comparada com a saída esperada. Se forem iguais, `corretos` recebe mais 1. Depois da iteração, a taxa de acerto é calculada dividindo o número de resultados corretos pelo número total de entradas, retornando em seguida três valores, `corretos` (número de acertos da rede), o número total de entradas e o `percentual` (taxa de acerto).

A complexidade dessa função é O(n * p), onde *n* é o número de entradas de teste e *p* é o número de camadas, devido à necessidade de calcular as saídas e comparar os resultados para cada entrada.

2.3 Classe Camada | `layer.py`

A classe é responsável por representar uma camada da rede neural, seja a camada de entrada, oculta ou saída. Cada camada é composta por vários neurônios, que têm a função de processar as entradas e gerar saídas que serão passadas para a próxima camada ou, no caso da última camada, usadas como resposta final da rede.

```python
class Camada:
    def __init__(self, camadaAnterior, numNeuronios, taxaAprendizagem, funcaoAtivacao, derivadaAtivacao):
        self.camadaAnterior = camadaAnterior
        self.neuronios = []


        for _ in range(numNeuronios):
            if camadaAnterior is None:
                pesos = []
            else:

                pesos = [random.uniform(-0.5, 0.5) for _ in range(len(camadaAnterior.neuronios))]


            neuronio = Neuronio(len(camadaAnterior.neuronios) if camadaAnterior else 0, taxaAprendizagem, funcaoAtivacao, derivadaAtivacao)
            self.neuronios.append(neuronio)


        self.saidasCache = [0.0 for _ in range(numNeuronios)]

    def calcularSaidas(self, entradas):
        if self.camadaAnterior is None:
            self.saidasCache = entradas
        else:
            self.saidasCache = [n.calcularSaida(entradas) for n in self.neuronios]
        return self.saidasCache
```

Figura 5 - Representação dos dois primeiros métodos da classe Camada

2.3.1 Inicialização da Camada | `__init__()`

O método inicializa a camada, criando os neurônios que compõem a camada e conectando eles à camada anterior. Durante a inicialização, são recebidos parâmetros como o número de neurônios, a função de ativação e a taxa de aprendizado.

A classe `Camada` cria uma lista de neurônios, onde cada neurônio é criado com base na quantidade de entradas fornecidas pela camada anterior, caso a camada anterior seja *None* (quando é camada de entrada), a camada não precisa de pesos, apenas de neurônios.

Para as camadas seguintes, os pesos dos neurônios são inicializados de forma aleatória, com base no número de neurônios da camada anterior, o que permite que a rede comece o treinamento com pesos distintos.

A complexidade dessa função é O(n * m), é a mesma da retropropagação, onde *n* é o número de neurônios e *m* é o número de entradas para cada neurônio.

2.3.2 Cálculo das Saídas da Camada | `calcularSaidas()`

O método é responsável por calcular as saídas de cada neurônio da camada, propagando os dados de entrada através da rede. Ele utiliza o método `calcularSaida()` de cada neurônio para calcular as saídas, aplicando a função de ativação.

Aqui, basta reforçar que o cálculo das saídas é feito com base nas entradas recebidas e que as saídas da camada são armazenadas na variável `saidasCache`. O fluxo de dados segue a ordem da rede, passando por cada camada até a útlima.

A complexidade dessa função também é O(n * m), já que o cálculo da saída de cada neurônio é baseado nas entradas e pesos.

2.3.3 Deltas na Camada de Saída | `calcularDeltasSaida()`

É responsável por calcular o erro (delta) para cada neurônio na camada de saída. Recebe apenas um parâmetro, que são as saídas desejadas `esperados`. Para cada neurônio na camada de saída, o delta é calculado como a diferença entre a saída gerada e a saída esperada, multiplicado pela derivada da função de ativação.

A complexidade dessa função é O(n), onde *n* é o número de neurônios na camada de saída, pois o cálculo do delta para cada neurônio envolve uma operação simples de subtração e multiplicação.

```python
def calcularDeltasSaida(self, esperados):
    for i in range(len(self.neuronios)):
        self.neuronios[i].delta = self.neuronios[i].derivadaAtivacao(self.neuronios[i].saidaCache) * (esperados[i] - self.saidasCache[i])

def calcularDeltasOculta(self, proximaCamada):
    for idx, neuronio in enumerate(self.neuronios):
        pesosProx = [n.pesos[idx] for n in proximaCamada.neuronios]
        deltasProx = [n.delta for n in proximaCamada.neuronios]
        soma = sum(p * d for p, d in zip(pesosProx, deltasProx))
        neuronio.delta = neuronio.derivadaAtivacao(neuronio.saidaCache) * soma
```

Figura 6 - Os dois últimos métodos restantes de *layer.py*

2.3.4 Deltas nas Camadas Ocultas | `calcularDeltasOculta()`

Calcula os deltas para os neurônios das camadas ocultas, utilizando os erros da camada seguinte (geralmente a camada de saída). O método recebe um parâmetro, que é a camada seguinte à camada atual, `proximaCamada`.

Para cada neurônio da camada oculta, o erro é calculado com base nos pesos e deltas da camada seguinte. A soma ponderada dos pesos e deltas da próxima camada é utilizada para calcular o delta da camada oculta, então esse erro (delta) é usado para ajustar os pesos da camada oculta.

A complexidade dessa função é O(n * m), onde *n* é o número de neurônios na camada e *m* o número de neurônios na próxima camada, pois o cálculo de deltas envolve a iteração sobre os neurônios de ambas as camadas.

2.4 Classe Neurônio | `neuron.py`

A classe `Neuronio`, representa a unidade básica de processamento dentro da rede neural. O comportamento do neurônio artificial pode ser melhor compreendido no início deste relatório. Por aqui, vamos seguir com a explicação da parte lógica por trás da classe.

```python
class Neuronio:
    def __init__(self, numEntradas, taxaAprendizagem, funcaoAtivacao, derivadaAtivacao):
        # Inicialização dos pesos de forma aleatória entre -0.5 e 0.5
        self.pesos = [random.uniform(-0.5, 0.5) for _ in range(numEntradas)]  # Pesos aleatórios
        self.funcaoAtivacao = funcaoAtivacao
        self.derivadaAtivacao = derivadaAtivacao
        self.taxaAprendizagem = taxaAprendizagem
        self.saidaCache = 0.0
        self.delta = 0.0

    def calcularSaida(self, entradas):
        # Calculando a saída do neurônio
        self.saidaCache = produtoEscalar(entradas, self.pesos)
        return self.funcaoAtivacao(self.saidaCache)
```

Figura 7 - Representação visual da classe Neuronio.

2.4.1 Inicialização do Neurônio | `__init__()`

O método é o construtor da classe e é responsável por inicializar o neurônio, configurando seus pesos, função de ativação e a taxa de aprendizado. Recebe como parâmetros os `pesos`, valores que ajustam a importância das entradas, `taxaAprendizagem`, que é a taxa com que os pesos serão ajustados durante o treinamento da rede, por fim, `funcaoAtivacao` que é a função usada para calcular a saída do neurônio e a sua derivada `derivadaAtivacao`, usada para ajustar os pesos durante o processo de retropropagação.

A classe recebe seus pesos e atributos de configuração, e armazena esses valores. Esses pesos serão atualizados durante o treinamento, à medida que a rede aprende com os dados. Então, inicializa o delta (erro) e a `saidaCache` (saída do neurônio).

A complexidade dessa função é O(1), pois a inicialização de um neurônio é uma operação simples de atribuição de valores para os pesos e funções de ativação.

2.4.2 Cálculo da Saída do Neurônio | `calcularSaida()`

Semelhante ao cálculo de saídas que explicamos anteriormente, o método calcula a saída exclusivamente de um neurônio. Primeiro, recebe as entradas do neurônio e as multiplica pelos pesos correspondentes, então temos a soma ponderada. O produto escalar das entradas e pesos é calculado e

o resultado dessa soma é armazenado em `saidaCache`. Em seguida, o resultado é passado pela função de ativação, que transforma o valor em uma saída.

A complexidade dessa função é O(m), onde *m* é o número de entradas, pois a função realiza uma soma ponderada das entradas e aplica a função de ativação.

2.5 Funções Auxiliares | `util.py`

Aqui, alguns conceitos matemáticos podem ficar mais claros, pois o arquivo contém funções auxiliares essenciais. Essas funções são chamadas em várias partes do código, como na classe `Neuronio` e `Camada`, e são responsáveis por realizar operações matemáticas e de processamento de dados necessárias para o funcionamento da rede.

O **produto escalar** calcula a soma ponderada entre as entradas de um neurônio e seus pesos. Ele é usado para determinar a contribuição de cada entrada no cálculo da saída do neurônio. A operação é simples e fundamental para o cálculo das saídas. A complexidade dessa função é O(n), onde *n* é o número de elementos nos vetores de entrada, pois a operação envolve uma soma ponderada simples."

```python
def produtoEscalar(xs, ys):
    return sum(x * y for x, y in zip(xs, ys))

def sigmoide(x):
    return 1 / (1 + exp(-x))

def derivadaSigmoide(x):
    s = sigmoide(x)
    return s * (1 - s)

def normalizarPorCaracteristica(dataset):
    for col in range(len(dataset[0])):
        colVals = [row[col] for row in dataset]
        minVal, maxVal = min(colVals), max(colVals)
        for row in dataset:
            row[col] = (row[col] - minVal) / (maxVal - minVal)
```

Figura 8 - Funções auxiliares cruciais nos cálculos da rede neural.

A próxima função do nosso módulo, já fizemos algumas menções ao decorrer do relatório, é a **sigmoide**, função de ativação muito utilizada em redes neurais. Ela transforma o vetor de entrada em um valor entre 0 e 1. Isso é feito pela fórmula:

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

A função sigmoide ajuda a introduzir não linearidade na rede, permitindo que ela aprenda padrões mais complexos, como explicado anteriormente.

Já que falamos da função de ativação, também temos que falar de sua derivada, pois é necessária para a retropropagação, ela é usada para calcular o

gradiente do erro em relação aos pesos. A derivada da função sigmoide é dada por:

$$\text{derivadaSigmoide}(x) = sigmoide(x) \times (1 - sigmoide(x))$$

Essa função é usada durante o cálculo da retropropagação para ajustar os pesos da rede. A complexidade dessas funções é O(1), pois elas realizam operações aritméticas simples para calcular o valor da função de ativação e sua derivada.

Por último, a função `normalizarPorCaracteristica()` tem como objetivo normalizar os dados de entrada, ajustando os valores de cada característica (ex. cada coluna de um vetor de entrada) para uma faixa padrão. Embora o conceito de normalização possa inicialmente parecer difícil de entender, especialmente para quem está começando, ela é fundamental para garantir que os dados de entrada tenham uma escala uniforme. Isso é importante porque dados com escalas muito diferentes podem dificultar o treinamento da rede neural, e a normalização ajuda a melhorar a convergência durante o processo de aprendizado.

A complexidade dessa função é O(n * m), onde n é o número de amostras e m o número de características, já que a função percorre cada valor das características para normalizar os dados.

2.6 Carregamento de Imagens | dataset_imagem.py

```python
def imagemParaVetor(path):
    img = Image.open(path).convert('L').resize((10, 10))
    return [1 if p < 128 else 0 for p in img.getdata()]  # 1=preto, 0=branco

def carregarDatasetImagens(pastaBase):
    X, y = [], []
    rotulos = {'quadrado':[1,0,0], 'circulo':[0,1,0], 'triangulo':[0,0,1]}
    for classe in rotulos.keys():
        pasta = os.path.join(pastaBase, classe)
        for nome_arquivo in os.listdir(pasta):
            if nome_arquivo.endswith('.png'):
                caminho = os.path.join(pasta, nome_arquivo)
                X.append(imagemParaVetor(caminho))
                y.append(rotulos[classe])
    return X, y
```

Figura 9 - Funções do módulo que carrega as imagens

O arquivo é responsável por carregar as imagens de treinamento que serão utilizadas pela rede neural. A principal função desse arquivo é transformar imagens em vetores de entrada, que a rede pode usar durante o treinamento. Vamos entender melhor função por função:

2.6.1 Conversão de Imagem em Vetor | imagemParaVetor()

A função que é usada como uma auxiliar no módulo, pode ser observada na **Figura 9**, ela recebe uma imagem como parâmetro, e converte em um vetor de dados numéricos. Esse vetor pode ser usado como entrada para a rede neural.

A complexidade dessa função é O(n), onde *n* é o número de pixels na imagem, pois a função percorre todos os pixels para converter a imagem em um vetor de dados.

2.6.2 Carregar Imagens | `carregarDatasetImagens()`

Esse método tem como objetivo carregar as imagens da pasta `data/imagens/`, onde estão armazenadas. Ele utiliza a função que explicamos anteriormente, para converter cada imagem em um vetor.

Primeiro, ela percorre as imagens. Para cada uma delas, a função `carregarImagem()` é chamada, então a função `imagemParaVetor()` converte a imagem em um vetor de dados. Também coleta as classificações das imagens, que representam as formas geométricas associadas a cada imagem. Por último, a função retorna as imagens e suas classificações, que são utilizados para o treinamento. A complexidade dessa função é O(n * m), onde *n* é o número de imagens e *m* é o número de pixels por imagem, pois a função carrega e processa cada imagem da pasta.

2.7 Gerar Imagens | `gerar_imagem.py`

```python
def salvarImagemQuadrado(nome):
    img = Image.new('L', (10, 10), color=255)
    draw = ImageDraw.Draw(img)
    draw.rectangle([2,2,7,7], fill=0)  # quadrado preto centralizado
    img.save(nome)

def salvarImagemCirculo(nome):
    img = Image.new('L', (10, 10), color=255)
    draw = ImageDraw.Draw(img)
    draw.ellipse([2,2,7,7], fill=0)   # círculo preto centralizado
    img.save(nome)

def salvarImagemTriangulo(nome):
    img = Image.new('L', (10, 10), color=255)
    draw = ImageDraw.Draw(img)
    draw.polygon([ (5,2), (2,7), (7,7) ], fill=0) # triângulo preto
    img.save(nome)
```

Figura 10 - Três funções que salvam formas geométricas diferentes.

2.7.1 Salvar Imagem | Quadrado, Círculo e Triângulo

Vamos resumir as três primeiras funções do módulo, que podem ser observadas na **Figura 10**. Essas três funções, tem o objetivo de criar e salvar as imagens de formas geométricas. Essas formas são geradas em uma imagem de 10x10 pixels, utilizando a biblioteca *PIL*, e são salvas na pasta `data/imagens/` para utilização no treinamento. As funções salvam as imagens com os nomes correspondentes à forma geométrica (e.g. quadrado.png).

A complexidade dessas funções é O(1), pois a operação de salvar uma imagem em disco é uma operação constante e não depende de variáveis externas.

```python
def gerarTodas():
    os.makedirs("data/imagens/quadrado", exist_ok=True)
    os.makedirs("data/imagens/circulo", exist_ok=True)
    os.makedirs("data/imagens/triangulo", exist_ok=True)
    # mudar range, gera +
    for i in range(10):
        salvarImagemQuadrado(f"data/imagens/quadrado/q_{i}.png")
        salvarImagemCirculo(f"data/imagens/circulo/c_{i}.png")
        salvarImagemTriangulo(f"data/imagens/triangulo/t_{i}.png")
```

Figura 11 - Função que gera as formas geométricas

2.7.2 Gerar Todas as Imagens | `gerarTodas()`

Para finalizar o módulo, a função é responsável por gerar e salvar múltiplas imagens de cada uma das formas geométricas. Ela faz o chamado das três últimas funções para criar 10 imagens de cada forma, salvando nas pastas. Pode-se observar também que antes da iteração, através da função `makedirs()`, que faz parte da biblioteca `os`, criamos uma pasta para cada forma.

A complexidade dessa função é O(n), onde *n* é o número de imagens geradas, pois a função salva um conjunto fixo de imagens para cada tipo de forma.

2.8 Casos Limítrofes | `limitrofes.py`

Para enriquecer ainda mais nosso conteúdo, vamos introduzir casos limítrofes! Esses casos são representações híbridas entre as formas geométricas originais que vamos trabalhar no projeto.

Antes de explorarmos mais, é importante que você visualize as formas geométricas (triângulo, quadrado e círculo), para entender os limites dessa variação. **Na Figura 12**, você pode observar as formas clássicas, que servirão como base para a criação dos casos limítrofes.

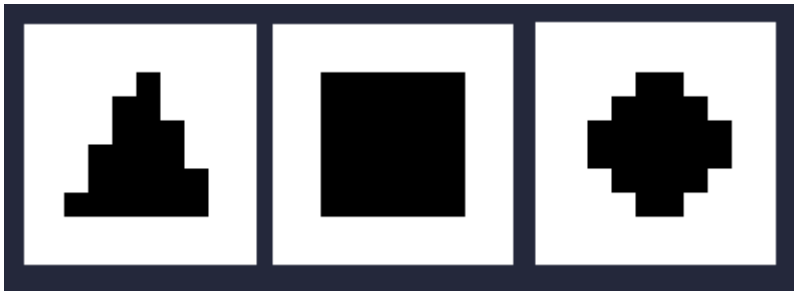

Figura 12 - Triângulo, quadrado e círculo. Formas geométricas 10x10 pixeladas que servem de dados. (As imagens estão com 1000% de zoom)

Visto as formas geométricas, vamos explorar as imagens híbridas. Essas imagens representam transições suaves entre as formas originais, desafiando a rede neural a identificar formas que não pertencem completamente a uma única categoria, mas sim a mistura delas.

Na **Figura 13**, você pode observar imagens híbridas, que são criadas a partir de formas clássicas mostradas na **Figura 12**.

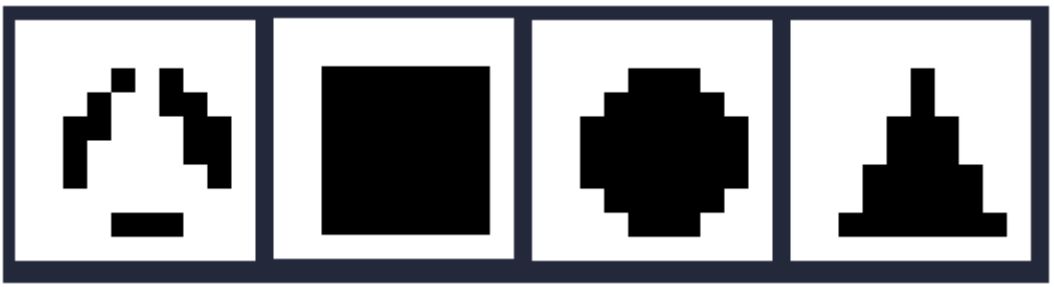

Figura 13 - Imagens híbridas geradas pela função gerarImagensLimitrofes(). (As imagens estão com 1000% de zoom)

Você pode notar que as formas não são perfeitamente definidas. A rede neural, ao processar essas imagens, precisa decidir qual forma geométrica ela mais se aproxima, fazendo isso através da **probabilidade**. Para cada imagem, a rede calcula a probabilidade dela pertencer a uma das formas geométricas, e a forma com maior probabilidade é a escolhida como a classificação final. A lógica por trás dessa classificação é identificar a forma que predomina, ou seja, qual forma tem mais características visíveis.

Agora, podemos entender como o arquivo `limitrofes.py` cria essas transições. Este arquivo possui funções responsáveis por gerar as imagens limítrofes e testá-las.

2.8.1 Gerando Imagens | `gerarImagensLimitrofes()`

```python
def gerarImagensLimitrofes():

    print("\nGerando mais imagens limitrofes ... \n")

    os.makedirs('data/imagens/limitrofe', exist_ok=True)

    for i in range(7, 13):
        img = Image.new('L', (10, 10), color=255)
        draw = ImageDraw.Draw(img)

        if i == 7:
            draw.rectangle([2, 2, 8, 8], fill=0, outline=0, width=2)
            img.save(f'data/imagens/limitrofe/quadrado_circulo_4.png')
            print(f"Imagem salva: quadrado_circulo_4.png")
```

Figura 14 - Função responsável por gerar as imagens limítrofes. Vale lembrar que existem mais variações, mas na representação da função só é possivel ver a primeira condição.

A função é responsável pela criação das imagens híbridas. Ela gera diversas variações entre quadrados, círculos e triângulos. O objetivo é criar uma interseção suave entre as formas, de forma que a rede neural seja desafiada a reconhecer a forma predominante, mesmo quando as imagens estão parcialmente misturadas.

Esse processo cria 6 novas imagens limítrofes. As imagens são geradas em diferentes combinações, como quadrado-círculo, triângulo-quadrado e círculo-triângulo. Você pode notar, por exemplo, que na **Figura 13**, a primeira imagem representa um círculo-triângulo, enquanto a segunda imagem representa um quadrado-círculo.

2.8.2 Testando Imagens | `testarLimitrofes()`

```python
def testarLimitrofes():

    img_paths = [
        "data/imagens/limitrofe/quadrado_circulo_1.png",
        "data/imagens/limitrofe/quadrado_circulo_2.png",
        "data/imagens/limitrofe/quadrado_circulo_3.png",
        "data/imagens/limitrofe/triangulo_quadrado_1.png",
        "data/imagens/limitrofe/triangulo_quadrado_2.png",
        "data/imagens/limitrofe/triangulo_quadrado_3.png",
    ]

    print("\n----------------------------------------------------")
    print("Iniciando o teste com imagens limitrofes ... \n")


    rede = RedeNeural([100, 15, 3], 0.3)

    for img_path in img_paths:

        if not os.path.exists(img_path):
            print(f"Erro: A imagem {img_path} não foi encontrada.")
            continue

        nome_legivel = imagem_descricao[os.path.basename(img_path)]

        print(f"\nProcessando a imagem limitrofe: {nome_legivel}")
        img = Image.open(img_path).convert('L').resize((10, 10))
        img_vector = [1 if p < 128 else 0 for p in img.getdata()]


        predicao = rede.calcularSaidas(img_vector)

        predicao_legivel = interpretarSaida(predicao)
        esperado = "quadrado"
        acertou = "ACERTOU!" if predicao_legivel == esperado else "ERROU."

        print(f"Probabilidades: {predicao}")
        print(f"Esperado: {esperado} | Predito: {predicao_legivel} - {acertou}")

    print("\n----------------------------------------------------\n")
```

Figura 15 - Função que faz a avaliação dos casos limítrofes.

Função responsável por processar e testar as imagens criadas pela função anterior. Ela realiza a avaliação dessas imagens pela rede neural, verificando se a rede consegue classificar corretamente as imagens híbridas.

Para isso, ela carrega as imagens limítrofes, converte cada uma delas em um formato compatível (vetores binários) para ser processada pela rede e, em seguida, utiliza a rede neural para fazer a classificação. Para cada imagem, a função calcula as probabilidades de pertencimento a cada forma geométrica e escolhe a forma com maior probabilidade como a predição final (**Figura 15**). Os resultados para ver se a rede conseguiu identificar as imagens serão apresentados no executor da rede, nos próximos tópicos.

Nota importante sobre as probabilidades:

Ao classificar cada imagem, a rede gera um vetor de probabilidades representando o grau de confiança em cada classe (triângulo, círculo e quadrado). Por exemplo, o vetor [0.69, 0.36 e 0.67] indica que a rede "acredita" em 69% que é um quadrado, 36% círculo e 67% triângulo. A classe escolhida é sempre aquela com maior valor de probabilidade. Esse valor reflete o quanto a rede está "segura" em sua predição e, em casos limítrofes, as probabilidades podem ser próximas, indicando que a rede está indecisa ao escolher a classe final.

2.9 Treinamento da Rede | `treinar_imagem.py`

As funções presentes neste arquivo são responsáveis por treinar a rede neural utilizando os dados de imagens que foram gerados e carregados anteriormente. A principal é a função `treinarRedeImagens()`, e temos no mesmo arquivo, como auxiliar, a função `interpretarSaida()`.

```python
def treinarRedeImagens():

    X, y = carregarDatasetImagens("data/imagens")
    temp = list(zip(X, y))
    random.shuffle(temp)
    X, y = zip(*temp)
    X, y = list(X), list(y)

    n_treino = int(0.8 * len(X))  # 80% treino
    X_treino, y_treino = X[:n_treino], y[:n_treino]
    X_teste, y_teste = X[n_treino:], y[n_treino:]

    vetor_para_nome = {(1,0,0):"quadrado", (0,1,0):"circulo", (0,0,1):"triangulo"}
    nomes_esperados = [vetor_para_nome[tuple(label)] for label in y_teste]

    print("Iniciando treinamento da rede neural para reconhecer formas geométricas em imagens 10×10.")
    print(f"Serão usados {len(X_treino)} imagens para treino e {len(X_teste)} para teste.")
    print("O programa irá mostrar o esperado, o predito e se a rede acertou cada imagem de teste.\n")

    rede = RedeNeural([100, 15, 3], 0.3)
    for _ in range(200):
        rede.treinar(X_treino, y_treino)

    corretos = 0
    print("Resultados dos testes:")
    for i, (entrada, esperado) in enumerate(zip(X_teste, nomes_esperados)):
        saida = rede.calcularSaidas(entrada)
        predito = interpretarSaida(saida)
        certo = "ACERTOU!" if predito == esperado else "ERROU."
        print(f"Teste {i+1:02d}: Esperado: {esperado:9s} | Predito: {predito:9s} {certo}")
        if predito == esperado:
            corretos += 1

    total = len(X_teste)
    percentual = corretos / total * 100

    print(f"\nA rede neural acertou {corretos} de {total} imagens de teste ({percentual:.2f}%).")
    if percentual ⩾ 80:
        print("Ótimo desempenho! A rede está generalizando bem.")
    elif percentual ⩾ 50:
        print("Desempenho razoável, mas pode melhorar com mais dados ou mais treino.")
    else:
        print("A rede ainda está aprendendo. Experimente aumentar as épocas ou gerar mais exemplos.")

    resposta = input("\nDeseja rodar o teste novamente? (s/n): ").strip().lower()
    if resposta == 's':
        print("\nReiniciando teste...\n")
        treinarRedeImagens()
    else:
        print("Encerrando programa.")
```

Figura 16 - Bloco completo da função *treinarRedeImagens()*

A função é responsável por orquestrar o treinamento da rede. Ela carrega o dataset de imagens e saídas esperadas. Para cada entrada, ela chama o método `treinar()`, alimentando a rede com os dados e ajustando os pesos com base nos erros calculados.

Além disso, ela avalia a rede com base nas imagens de teste, comparando as saídas previstas com as saídas esperadas. Para isso, chama o método `calcularSaidas()` para calcular a saída da rede para cada entrada de teste. Então, a saída gerada é interpretada por meio da função `interpretarSaida()`, que converte a saída numérica da rede em uma categoria legível.

Ela também mantém um contador de acertos `corretos`, que vimos na função `validar()`, permitindo calcular a taxa de acerto da rede ao final do teste. O desempenho é avaliado, e um feedback é dado, indicando o quão bem a rede aprendeu a classificar as formas.

Caso a precisão seja baixa, o código sugere melhorias, como aumentar as épocas de treinamento ou gerar mais exemplos de treino. A função também permite ao usuário continuar o treinamento para ajustar os parâmetros, ou encerrar o processo caso o desempenho seja satisfatório.

A complexidade dessa função é O(k * n * m), onde *k* é o número de épocas, *n* o número de imagens de treinamento, e *m* o número de neurônios,

pois a função realiza o treinamento da rede sobre cada imagem e em cada camada da rede.

3.0 Executando a Rede Neural | `main.py`

Após uma explicação detalhada sobre o funcionamento da rede, chegamos à parte da execução e avaliação dos resultados. Nesta etapa, o programa realiza dois tipos de testes: um para as formas geométricas convencionais e outro para os casos limítrofes, que desafiam a capacidade da rede em identificar formas intermediárias entre as originais.

3.0.1 Execução com Formas Geométricas Convencionais

Nesse processo, a rede neural é testada com imagens das formas triângulo, círculo e quadrado, e a precisão da rede é medida. Para cada imagem de teste, o programa exibe os resultados comparando o valor esperado com a predição da rede, além de calcular a taxa de acerto. Isso oferece uma visão clara de como a rede está desempenhando em relação às formas geométricas clássicas.

Os resultados dos testes são apresentados, destacando a precisão da rede na classificação das imagens, com a rede **acertando 100% dos casos**! Logo depois, o usuário tem a opção de rodar o teste novamente.

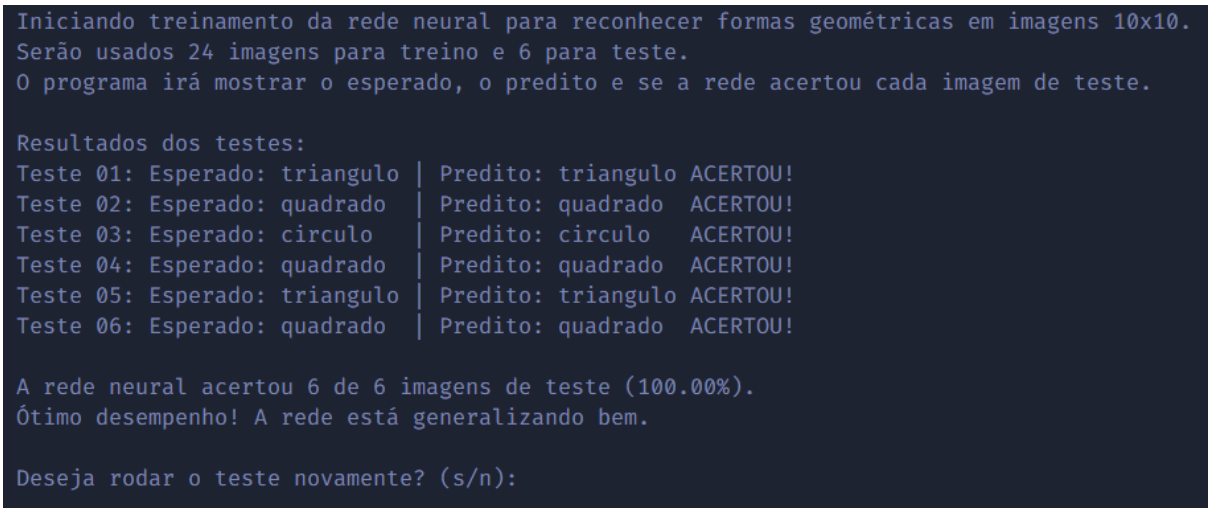

Figura 17 - Resultados da execução das formas geométricas clássicas.

Apesar dos resultados serem satisfatórios, inicialmente tive problemas. A rede não acertava mais de 50%, então tive que fazer alguns ajustes que são comentados em Desafios Enfrentados.

3.0.2 Execução com casos Limítrofes

Então, realizamos os testes com os casos limítrofes. Nesse estágio, a rede é desafiada a identificar imagens híbridas, como quadrado-círculo, triângulo-quadrado e círculo-triângulo, que são misturas das formas geométricas originais.

Durante os primeiros testes, os resultados não foram agradáveis: a rede neural frequentemente errava a classificação dessas imagens. Essa dificuldade se deve pela rede não ter sido exposta aos casos limítrofes durante o treinamento, só reconhecendo formas puras.

```
Iniciando o teste com imagens limitrofes...


Processando a imagem limitrofe: Quadrado-Círculo 1
Probabilidades: [0.3558756554362665, 0.7110472828318883, 0.7979379485084376]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Quadrado-Círculo 2
Probabilidades: [0.3558756554362665, 0.7110472828318883, 0.7979379485084376]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Quadrado-Círculo 3
Probabilidades: [0.3558756554362665, 0.7110472828318883, 0.7979379485084376]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Triângulo-Quadrado 1
Probabilidades: [0.41835710081228394, 0.7733221501369822, 0.8116089127742643]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Triângulo-Quadrado 2
Probabilidades: [0.41835710081228394, 0.7733221501369822, 0.8116089127742643]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Triângulo-Quadrado 3
Probabilidades: [0.41835710081228394, 0.7733221501369822, 0.8116089127742643]
Esperado: quadrado | Predito: triangulo - ERROU.
```

Figura 18 - Dificuldades enfrentadas ao executar casos limítrofes

Após essa analise, realizei duas modificações que foram fundamentais: primeiro, incluí algumas imagens híbridas no conjunto de treinamento, rotulando elas de acordo com a predominância visual; em seguida, aumentei o número de **épocas de treino** (em `treinar_imagem.py`) para que a rede tivesse mais oportunidades de ajustar seus parâmetros.

Com esses ajustes, a rede passou a generalizar melhor, porém percebi que o valor “esperado” nos testes de imagens limítrofes estavam sendo exibidos de forma incorreta, aparecendo sempre como “quadrado”,

independentemente da classe real da imagem testada. Isso ocorria porque o código anterior utilizava um valor fixo para o rótulo esperado, ao invés de associá-lo dinamicamente a cada arquivo de imagem.

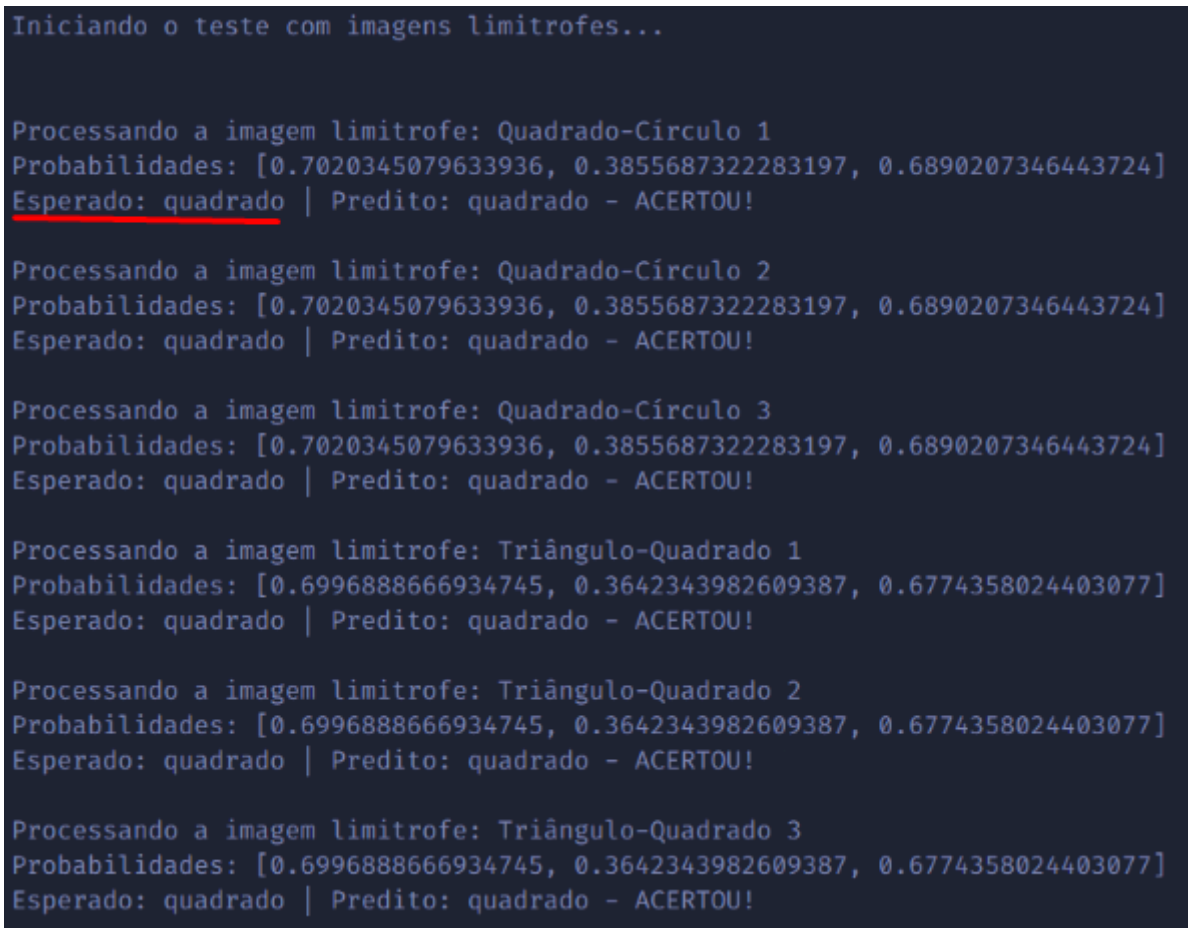

Figura 19 - Após os reajustes, ao executar a rede, o valor “esperado” não era fiel à imagem.

Para corrigir o problema, foi criado um dicionário de mapeamento entre o nome do arquivo de cada imagem limítrofe e sua classe correta. Com isso, o teste passou a exibir o valor esperado correto, tornando “Esperado” mais fiel. Na imagem a seguir, você pode observar que apesar do problema estar fixado, a rede ainda não acertava 100% das vezes.

```
Processando a imagem limitrofe: Quadrado-Círculo 1
Probabilidades: [0.5408703652156535, 0.39773622844220985, 0.6881982189224803]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Quadrado-Círculo 2
Probabilidades: [0.5408703652156535, 0.39773622844220985, 0.6881982189224803]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Quadrado-Círculo 3
Probabilidades: [0.5408703652156535, 0.39773622844220985, 0.6881982189224803]
Esperado: quadrado | Predito: triangulo - ERROU.

Processando a imagem limitrofe: Triângulo-Quadrado 1
Probabilidades: [0.5035627635208175, 0.4711800899229247, 0.6741627445557957]
Esperado: triangulo | Predito: triangulo - ACERTOU!

Processando a imagem limitrofe: Triângulo-Quadrado 2
Probabilidades: [0.5035627635208175, 0.4711800899229247, 0.6741627445557957]
Esperado: triangulo | Predito: triangulo - ACERTOU!

Processando a imagem limitrofe: Triângulo-Quadrado 3
Probabilidades: [0.5035627635208175, 0.4711800899229247, 0.6741627445557957]
Esperado: triangulo | Predito: triangulo - ACERTOU!
```

Figura 20 - O problema foi corrigido, mas a rede ainda não consegue acertar todas as vezes.

No entanto, podemos executar múltiplos ciclos de treinamento e teste. Então ao optar por rodar novamente, a rede neural é reinicializada, os dados são reembaralhados e todo o processo de treinamento é refeito. Na segunda execução, a rede frequentemente passa a acertar todos os casos limítrofes, demonstrando na prática, sua capacidade de aprendizado contínuo e adaptação.

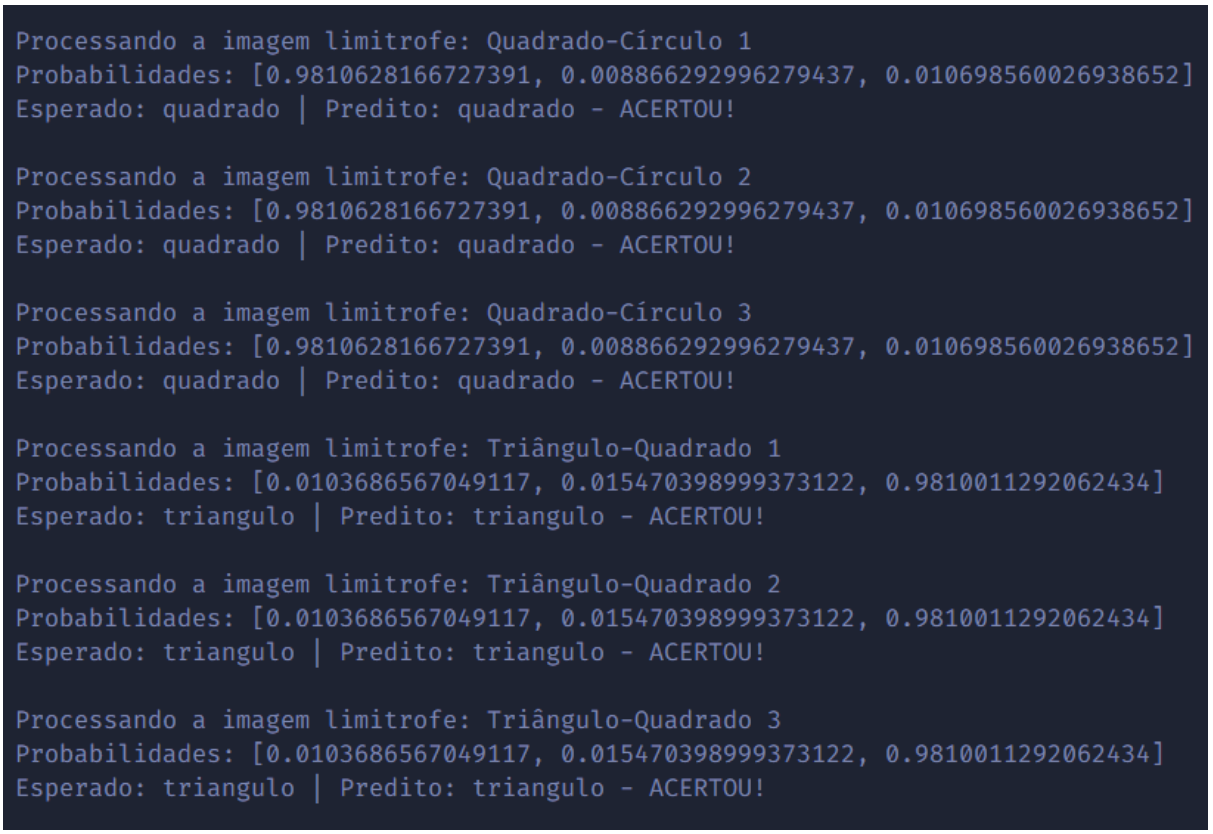

Figura 21 - Ao repetir os testes, a rede apresenta um desempenho notavelmente superior, alcançando 100% de acerto nos casos limítrofes.

4. Conclusão

4.1 Resumo do projeto

Neste projeto, desenvolvemos uma rede neural *feedforward* simples para reconhecimento de formas geométricas. A rede foi treinada utilizando um dataset de imagens, onde quadrados, círculos e triângulos foram classificados de forma correta pelo programa, também foram testados casos limítrofes. O foco foi na **aprendizagem supervisionada** (um tipo de aprendizado de máquinas em que o modelo é treinado com dados rotulados), onde a rede foi alimentada com entradas (que são as imagens) e saídas esperadas (a forma geométrica).

Para consolidar os conceitos mencionados no relatório, é de grande importância que o leitor faça consulta do livro.

4.2 Desafios Enfrentados

Os principais desafios do projeto foram obter bons resultados tanto nas imagens padrão quanto nos casos limítrofes. Inicialmente, a rede apresentava baixo desempenho em ambos: errava parte das formas clássicas e quase sempre falhava nas imagens híbridas. Após ajustar taxa de aprendizado, aumentar o número de épocas e incluir exemplos limítrofes no treinamento, a rede passou a generalizar melhor, alcançando 100% de acerto nos testes finais.

4.3 Aprendizado com a Construção

Este foi um projeto que eu desenvolvi enquanto estudava sobre o tema, que também foi um desafio enfrentado, fiz consultas no livro mencionado na introdução, que foi usado como base teórica e até prática. Foram 7 dias de estudos constantes. Apesar disso, consegui aprender conceitos-chave, que serão fundamentais para meus estudos. Esse projeto me proporcionou uma experiência de muito valor, dando base para seguir em direção ao domínio da inteligência artificial.