

# Implementação de uma Rede Neural para Reconhecimento de Dígitos Manuscritos em Imagens

Alex Seródio Gonçalves

Departamento de Sistemas e Computação  
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil  
{alexserodio}@furb.br

**Resumo.** *Este artigo apresenta a implementação de uma rede neural artificial capaz de reconhecer dígitos manuscritos em imagens de 28x28 pixels, implementada utilizando a linguagem de programação Octave. A rede desenvolvida utiliza a arquitetura feedforward e é composta por três camadas, onde cada neurônio utiliza a função sigmoide como função de ativação. Os resultados alcançados pela rede foram satisfatórios, possuindo uma taxa de, em média, 94% de precisão, o que mostra que a rede cumpre seu objetivo de reconhecer dígitos manuscritos em imagens.*

## 1. Introdução

O processamento e reconhecimento de imagens, por mais complexo que possa ser, é uma tarefa facilmente realizada por seres humanos. Isso é possível graças ao córtex visual primário e secundário, existente nos dois hemisférios do cérebro humano. Cada parte do córtex é dividido em cinco áreas, denominadas V1, V2, V3, V4 e V5, responsáveis pelo processamento e reconhecimento de complexos padrões recebidos através de estímulos visuais. Cada área possui milhões de neurônios, conectados entre si através de bilhões de conexões sinápticas que realizam processos cada vez mais complexos de reconhecimento (NIELSEN, 2017).

O aprendizado de máquina (*machine learning*) é uma área da inteligência artificial voltada ao desenvolvimento de algoritmos e técnicas computacionais que permitem ao computador o aprendizado, ou seja, que façam com que o computador consiga armazenar e reter conhecimento de maneira automática a partir de exemplos – aperfeiçoando, portanto, o seu desempenho em uma ou mais tarefas (GOLDSCHMIDT, 2010, p. 35). O reconhecimento de imagens é uma das muitas tarefas solucionáveis por um computador através do aprendizado de máquina.

Um paradigma muito utilizado na área de aprendizado de máquina é o das Redes Neurais Artificiais (RNAs). Tal paradigma baseia-se em modelos matemáticos e busca reproduzir computacionalmente o funcionamento dos neurônios biológicos no reconhecimento de padrões existentes em imagens. Para isso utiliza-se uma estrutura inspirada no córtex visual humano, onde os neurônios, que processam os dados da imagem, são divididos em várias camadas (como as várias áreas do córtex visual) e se comunicam entre si através de conexões (como as conexões sinápticas), sendo que cada camada lida com aspectos cada vez mais complexos do reconhecimento (NIELSEN, 2017).

A utilização de linguagens de programação de domínio científico (ou linguagens de outros domínios que possuem pacotes externos com funcionalidades semelhantes) é uma escolha recorrente ao implementar Redes Neurais Artificiais. Isso se deve ao fato de que tais linguagens são otimizadas para o uso de cálculos matemáticos, incluindo operações matriciais, que são características fundamentais na implementação de RNAs. Nesse contexto apresenta-se a linguagem de programação Octave (2018).

Octave é uma linguagem de programação científica de licença livre, escrita principalmente, mas não exclusivamente, por John W. Eaton, em 1993, e possui ferramentas extensivas para resolver problemas comuns de álgebra linear numérica, com foco principal na computação numérica de problemas lineares e não lineares. Foi originalmente concebida para substituir o uso de linguagens como Fortran na resolução de problemas de engenharia química na universidade de Wisconsin-Madison, tornando-se depois uma linguagem muito mais flexível (OCTAVE, 2018).

Entre as diversas aplicações de aprendizado de máquina, uma das mais conhecidas academicamente é a utilização de RNAs no reconhecimento de dígitos manuscritos. Tal aplicação é amplamente utilizada no meio acadêmico como forma de ensino sobre a estrutura e funcionamento de Redes Neurais Artificiais por se tratar de um problema solucionável através de várias abordagens diferentes, desde a mais simples até a mais complexa.

Diante do apresentado, este artigo apresenta a criação de um modelo de rede neural artificial implementado utilizando a linguagem de programação Octave, que seja capaz de reconhecer dígitos manuscritos entre 0 e 9 a partir de imagens.

A estrutura do artigo está dividida em três capítulos, sendo que no primeiro foi apresentado uma introdução sobre o tema, a linguagem de programação utilizada e o objetivo do projeto. O segundo capítulo trata de detalhar as técnicas e abordagens utilizadas no desenvolvimento do projeto e, por fim, o terceiro capítulo apresenta uma discussão sobre as conclusões finais e resultados alcançados a partir dos estudos realizados.

## **2. Desenvolvimento**

Este capítulo apresenta o desenvolvimento do projeto e está dividido em cinco seções. Na seção 2.1 são listados os requisitos funcionais do projeto. A seção 2.2 apresenta a especificação da arquitetura da rede desenvolvida. A seção 2.3 apresenta algumas das técnicas e funções matemáticas utilizadas no processo de classificação das imagens. A seção 2.4 detalha a implementação dos algoritmos utilizados. Por fim, a seção 2.5 apresenta as questões de operacionalidade do projeto, demonstrando as etapas e sequências de execução.

### **2.1. Requisitos Funcionais**

Os requisitos funcionais do projeto desenvolvido são:

- a) desenvolver uma rede neural que reconheça dígitos manuscritos em imagens;
- b) treinar a rede de forma a reconhecer os dígitos manuscritos;
- c) utilizar a linguagem Octave para implementar a rede neural;

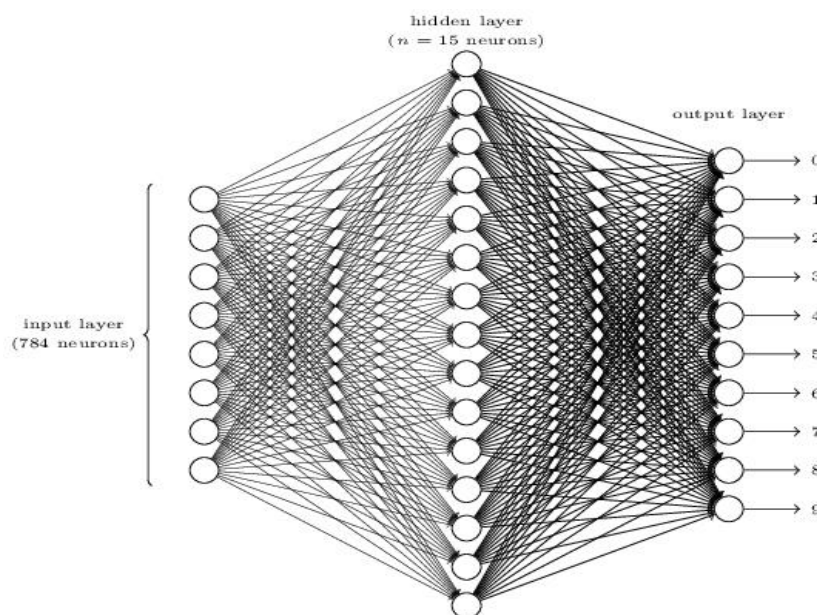
d) utilizar a base de dados MNIST no treinamento da rede.

## 2.2. Especificação

A arquitetura da rede neural desenvolvida é composta por três camadas, denominadas *input layer*, *hidden layer* e *output layer*. A primeira camada (*input layer*) trata de receber a imagem de entrada contendo o dígito a ser reconhecido. As imagens submetidas a rede possuem dimensões de 28 x 28 pixels, ou seja, 784 pixels, portanto a primeira camada é composta por 784 neurônios, sendo que cada um corresponde a um pixel da imagem.

A terceira camada (*output layer*) corresponde ao resultado alcançado pela rede neural. No problema proposto existem 10 possíveis resultados (dígitos entre 0 e 9) e, portanto, essa camada é composta por 10 neurônios, cada um representando um possível resultado. Isso ocorre pois, ao invés de a rede calcular o resultado exato, o que é calculado é a probabilidade de cada dígito ser o correto. Dessa forma, no final da execução do algoritmo, a camada terá 10 valores entre 0 e 1 indicando a probabilidade de cada dígito ser o correto, onde o maior valor representa o resultado mais provável, ou, em outras palavras, a hipótese alcançada pela rede.

Por fim, a segunda camada (*hidden layer*), diferentemente das outras duas, refere-se ao processamento intermediário existente entre a entrada e saída da rede e, portanto, não possui uma quantidade específica de neurônios. A quantidade de neurônios dessa camada é definida a partir de heurísticas predeterminadas ou testes realizados durante o desenvolvimento da rede, de forma a alcançar a melhor combinação entre tempo de processamento e precisão. No caso da arquitetura proposta, optou-se por utilizar 30 neurônios nesta camada, porém esse valor pode ser modificado (para mais ou menos neurônios), na tentativa de alcançar melhores resultados. O modelo de arquitetura proposta para a rede neural é mostrado na Figura 1<sup>1</sup>.



**Figura 1. Arquitetura da rede neural proposta**

<sup>1</sup> A quantidade de neurônios da primeira e segunda camada foi diminuída para facilitar a visualização.

Na Figura 1, além das três camadas e seus respectivos neurônios, é possível observar as conexões existentes entre cada neurônio de uma camada com todos os neurônios da camada seguinte. Essas conexões existentes entre pares de neurônios são chamadas de pesos sinápticos, que neste caso nada mais são do que valores reais que expressam a importância de cada neurônio (quanto maior o peso, maior a relevância do neurônio). Além disso, apesar de não ser visível na Figura 1, cada camada possui um valor a mais denominado bias, que controla a partir de que valor um neurônio específico será ativado (NIELSEN, 2017).

Computacionalmente, cada camada da rede é representada por um vetor vertical (uma matriz de uma coluna e n linhas), onde cada posição do vetor corresponde a um neurônio, que armazena um valor entre 0 e 1. Os pesos sinápticos, por sua vez, são representados por matrizes, onde a quantidade de linhas é o tamanho da camada seguinte e a quantidade de colunas é o tamanho da camada anterior. No caso da rede proposta, as camadas são representadas por três vetores de 784, 30 e 10 posições, respectivamente, onde cada camada possui um vetor de bias de mesmo tamanho, e os pesos são representados por duas matrizes com dimensões de 30x784 e 10x30. O chamado aprendizado de máquina consiste em encontrar os valores corretos para cada peso e bias da rede, de forma a alcançar a maior quantidade de hipóteses corretas possíveis.

### 2.3. Técnicas de classificação

Por utilizar o paradigma conexionista, onde as técnicas de aprendizado utilizam modelos matemáticos simplificados inspirados no modelo biológico do sistema nervoso (GOLDSCHMIDT, 2010, p. 36), e a arquitetura *feedforward*, onde a propagação das informações na rede ocorre sempre para a frente, faz-se necessário destacar algumas funções matemáticas importantes para o funcionamento da rede, sendo elas as funções *feedforward* e *sigmoid*.

A função *feedforward* calcula, a partir da camada atual, o valor dos neurônios da camada seguinte (daí o nome da função). Essa função é dada pelo produto matricial entre uma camada qualquer e seus respectivos pesos, somado com os bias da mesma camada, como mostrado no Quadro 1, onde  $W$  corresponde à matriz de pesos,  $a^1$  ao vetor de neurônios da camada atual,  $a^2$  ao vetor de neurônios da camada seguinte e  $b$  ao vetor de bias. Ao repetir esse processo para a primeira e segunda camada, a última camada conterá a hipótese alcançada pela rede.

**Quadro 1. Função *feedforward***

$$a^2 = Wa^1 + b$$

Após a aplicação da função *feedforward* em uma camada, antes de atribuir o resultado da função na camada seguinte, cada neurônio resultante da função *feedforward* deve ser submetido a uma função de ativação que, segundo Goldschmidt (2010, p. 79), “[...] determina o novo valor do estado de ativação deste neurônio, a partir de seu potencial de ativação.”. Alguns exemplos de possíveis funções de ativação utilizadas pelos neurônios seriam: função linear, gaussiana, degrau, tangente hiperbólica e

sigmoide (GUARDIA, 2003, p. 2). No caso da rede proposta, a função de ativação utilizada é a sigmoide, mostrada na Quadro 2, onde  $e$  é a constante matemática de Euler (tendo seu valor simplificado para 2.7183 no Octave) e  $x$  representa o valor submetido a função. O retorno dessa função, a partir de qualquer número submetido a ela, será sempre um valor entre 0 e 1.

**Quadro 2. Função *sigmoid***

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

É importante ressaltar que as funções apresentadas aqui estão lidando com as camadas e pesos da rede neural, que são representadas computacionalmente por vetores e matrizes. Por este motivo, as operações realizadas nessas estruturas seguem as regras impostas pela Álgebra Linear, onde qualquer operação realizada sobre uma das estruturas é na verdade realizada para cada elemento existente nela. Ou seja, ao submeter uma camada a qualquer uma das funções apresentadas aqui, na verdade estamos submetendo cada neurônio existente na camada à mesma função, e não apenas um único neurônio isolado. O mesmo ocorre para os vetores de bias e matrizes de pesos.

Além dessas, outras funções importantes, como a de *backpropagation*, serão apresentadas adiante, porém, essas não serão apresentadas através de modelos matemáticos, mas sim através de implementações de código fonte inspiradas em tais modelos.

## 2.4. Implementação

Para a implementação da rede neural desenvolvida foi utilizado a linguagem de programação Octave e para o treinamento utilizou-se a base de dados de imagens de dígitos manuscritos MNIST.

MNIST é uma base de dados composta por 70.000 imagens de dígitos manuscritos, separadas em um conjunto de 60.000 imagens utilizadas para o treinamento da rede e 10.000 imagens utilizadas para testa-la. Todas as imagens da base de dados possuem dimensões de 28x28 pixels, escritas com cor preta em fundo branco. Além disso, cada uma das 70.000 imagens possui um rótulo que corresponde ao número contido na imagem (LeCun et al, XXXX). Esses rótulos são importantes para validar se a hipótese alcançada pela rede durante o treinamento está correta ou não.

Cada conjunto de imagens está contido em um arquivo binário independente (*train-images.idx3-ubyte* para as imagens de treino e *t10k-images.idx3-ubyte* para as imagens de teste) e são lidos, estruturados e retornados pela função *loadMNISTImages*, apresentada no Quadro 3. Os conjuntos de rótulos também se encontram em arquivos binários independentes (*train-labels.idx1-ubyte* para os rótulos de treino e *t10k-labels.idx1-ubyte* para os rótulos de teste) e são lidos de forma semelhante, pela função *loadMNISTLabels*.

Na função apresentada no Quadro 3 é possível observar que o Octave utiliza variáveis com tipagem dinâmica, onde o tipo da variável é definido com base no valor

atribuído a ela, permitindo assim a criação de variáveis no meio do código sem a necessidade de defini-las previamente, além de possibilitar a alteração do tipo de uma variável através da atribuição de algum valor de tipo diferente para ela.

**Quadro 3. Leitura das imagens da base de dados MNIST**

1.	<code>function img = loadMNISTImages(filename)</code>
2.	<code>fp = fopen(filename, 'rb');</code>
3.	
4.	<code>numImages = fread(fp, 1, 'int32', 0, 'ieee-be');</code>
5.	<code>numRows = fread(fp, 1, 'int32', 0, 'ieee-be');</code>
6.	<code>numCols = fread(fp, 1, 'int32', 0, 'ieee-be');</code>
7.	<code>img = fread(fp, inf, 'unsigned char');</code>
8.	<code>img = reshape(img, numCols, numRows, numImages);</code>
9.	<code>img = permute(img,[2 1 3]);</code>
10.	
11.	<code>fclose(fp);</code>
12.	
13.	<code>img = reshape(img, size(img, 1) * size(img, 2), size(img, 3));</code>
14.	<code>img = double(img) / 255;</code>
15.	<code>end</code>

Após abrir o arquivo utilizando a função *fopen* na linha 2, utiliza-se nas linhas 4 a 6 a função *fread* para ler a quantidade de imagens contidas no arquivo e suas dimensões, passando como parâmetro para a função o arquivo, quantidade de dados a serem lidos, tipo do dado, intervalo entre as leituras e tipo da arquitetura do arquivo. Depois, nas linhas 7 a 9, todas as imagens são lidas e reestruturadas no formato de uma lista de matrizes, utilizando as informações adquiridas nas linhas 4 a 6. Por fim, nas linhas 13 e 14 as matrizes são normalizadas, de forma que seus valores passem a ser números entre 0 e 1.

A inicialização da rede e o treinamento da mesma são realizados pela função *start\_network\_training*, mostrada no Quadro 4. Aqui é interessante destacar a assinatura da função *start\_network\_training*. A linguagem Octave permite que funções retornem não apenas um único elemento, mas sim listas de elementos, sendo que os elementos retornados podem possuir o mesmo tipo ou tipos diferentes. Os elementos a serem retornados são especificados entre colchetes, logo após a palavra reservada *function*, e são retornados automaticamente após a função ser finalizada, sem a necessidade de utilizar uma palavra reservada como *return* para indicar o ponto de retorno (EATON, 2018).

Inicialmente, da linha 4 a 8, é feito a importação das imagens e rótulos de treino e teste contidas na base de dados MNIST, realizado através da função já apresentada no Quadro 3. Nas linhas 10 a 13, são definidas algumas informações importantes referentes a estrutura e especificação de treino da rede, sendo elas:

- *layers\_size*: uma lista de inteiros contendo o tamanho de cada uma das três camadas;
- *epochs*: indica a quantidade de seções de treino, isto é, a quantidade de vezes que as 60.000 imagens de treino serão submetidas para a rede. Quanto mais seções

de treino, maior a precisão da rede. Porém, ao utilizar valores maiores que 30, a precisão da rede tende a não sofrer grandes alterações e, portanto, não é recomendado utilizar valores maiores. Por outro lado, valores menores resultam em treinos mais rápidos, apesar de resultar em uma diminuição considerável de precisão;

- *batch\_size*: indica quantas imagens de teste serão submetidas a rede antes das alterações serem aplicadas aos pesos das camadas. No caso do *batch\_size* estar valendo 10 significa que, das 60.000 imagens, apenas as 10 primeiras serão inicialmente submetidas a função de aprendizado. A função de aprendizado irá calcular as modificações necessárias nos pesos de cada camada, as modificações serão aplicadas, e então o processo será repetido, utilizando as próximas 10 imagens da lista. Em outras palavras, essa variável define quantas vezes os pesos da rede serão atualizados durante o treinamento;
- *learning\_rate*: controla o quanto os pesos da rede devem ser ajustados a cada aplicação da função de aprendizado. Quanto menor o valor, menor as alterações aplicadas na rede e, conseqüentemente, com uma grande quantidade de dados de treino, mais preciso será as atualizações realizadas nos pesos. Esse valor também pode impactar a duração do treino.

**Quadro 4. Definição das variáveis de controle da rede e inicialização do treino**

1.	function [w1, w2, b2, b3] = start_network_training ()
2.	addpath('..../mnist_dataset');
3.	
4.	train_img = loadMNISTImages('..../mnist_dataset/train-images.idx3-ubyte');
5.	trainingY = loadMNISTLabels('..../mnist_dataset/train-labels.idx1-ubyte');
6.	
7.	test_img = loadMNISTImages('..../mnist_dataset/t10k-images.idx3-ubyte');
8.	test_values = loadMNISTLabels('..../mnist_dataset/t10k-labels.idx1-ubyte');
9.	
10.	layers_size = [784, 30, 10];
11.	epochs = 30;
12.	batch_size = 10;
13.	learning_rate = 3.0;
14.	
15.	[b2, b3, w1, w2] = stochastic_gradient_descent(train_img, train_values,
16.	layers_size, epochs, batch_size, learning_rate);
17.	
18.	result = validate_hipoteses(test_img, test_values, b2, b3, w1, w2);
19.	printf("Result: %d / %d\n", result, size(test_values, 1));
20.	endfunction

Em seguida, na linha 15, é chamada a função *stochastic\_gradient\_descent*, que recebe como parâmetro as imagens e rótulos de teste, junto com todas as variáveis de controle definidas anteriormente, e retorna os bias (*b2* e *b3*) e os pesos (*w1* e *w2*). Essa função é apresentada no Quadro 5 e deriva de uma função matemática de mesmo nome. Seu funcionamento consiste basicamente em dividir os dados de treino em conjuntos

menores e, a partir desses conjuntos, atualizar os pesos e bias da rede através de uma função auxiliar chamada *update\_network*.

**Quadro 5. Implementação do método do gradiente descendente**

1.	function [b2, b3, w1, w2] = stochastic_gradient_descent (trainX, trainY,
2.	layers_size, epochs, batch_size, learning_rate)
3.	
4.	batch_size -= 1;
5.	training_size = size(training_dataX, 2);
6.	
7.	b2 = randn(layers_size(2), 1);
8.	b3 = randn(layers_size(3), 1);
9.	w1 = randn(layers_size(2), layers_size(1));
10.	w2 = randn(layers_size(3), layers_size(2));
11.	
12.	for j = 1:epochs
13.	for i = 1:batch_size+1:training_size-batch_size
14.	batchX = trainX(:, [i:batch_size+i]);
15.	batchY = trainY(:, [i:batch_size+i]);
16.	[b2, b3, w1, w2] = update_network(batchX, batchY, learning_rate,
17.	b2, b3, w1, w2);
18.	endfor
19.	endfor
20.	endfunction

Nas linhas 7 a 10 é realizado a inicialização dos vetores de bias e matrizes de pesos com valores randômicos. A função *randn* utilizada aqui recebe como parâmetro as dimensões da matriz (quantidade de linhas e quantidade de colunas) e retorna uma matriz com essas dimensões contendo valores aleatórios entre 0 e 1.<sup>2</sup>

O primeiro laço de repetição utiliza a variável de controle *epochs* inicializada na função anterior para controlar a quantidade de seções de treino que serão realizadas. Para cada seção os dados de treino são divididos em conjuntos menores definidos pela variável *batch\_size*. Essa divisão ocorre dentro do segundo laço de repetição, nas linhas 9 e 10, onde as variáveis *batchX* e *batchY* recebem apenas as imagens contidas nas listas *trainX* e *trainY* no intervalo *i:batch\_size+i*, ou seja, desde a posição *i* até o próximo conjunto.

Em seguida esses dois conjuntos (*batchX* contendo as imagens e *batchY* contendo seus respectivos rótulos) são passados como parâmetro para a função *update\_network*. Essa função trata de chamar a função de *backpropagation* e aplicar as alterações calculadas por ela nos pesos e bias, como mostrado no Quadro 6.

---

<sup>2</sup> Essa inicialização de matrizes acontece em outras funções que serão apresentadas adiante, porém, serão ocultadas do código fonte a fim de mostrar apenas o trecho essencial das funções.



**Quadro 6. Atualização dos pesos e bias**

```
1. function [b2, b3, w1, w2] = update_network (batchX, batchY, learning_rate,
2.     b2, b3, w1, w2)
3.
4.     # inicialização das variáveis temp_w1, temp_w2, temp_b2 e temp_b3
5.     # com as mesmas dimensões dos pesos e bias, porém, valendo zero
6.
7.     batches = columns(batchX);
8.
9.     for x = 1:batches
10.        [d_b2, d_b3, d_w1, d_w2] = backpropagation(batchX(:,x), batchY(:,x),
11.            b2, b3, w1, w2);
12.
13.        temp_w1 += d_w1;
14.        temp_w2 += d_w2;
15.        temp_b2 += d_b2;
16.        temp_b3 += d_b3;
17.    endfor
18.
19.    w1 = w1 - ((learning_rate / batches) * temp_w1);
20.    w2 = w2 - ((learning_rate / batches) * temp_w2);
21.    b2 = b2 - ((learning_rate / batches) * temp_b2);
22.    b3 = b3 - ((learning_rate / batches) * temp_b3);
23. endfunction
```

A maior parte do trabalho feito aqui é realizado pela função de *backpropagation*, restando à função *update\_network* apenas a tarefa de aplicar as alterações calculadas por ela. A aplicação de tais alterações é realizada nas linhas 19 a 21, onde a soma das alterações é subtraída dos pesos e bias recebidos como parâmetro pela função. Porém, antes da subtração, esses valores são multiplicados pela variável *learning\_rate* dividido pela quantidade de conjuntos (*batches*). No fim, esses mesmos pesos e bias (agora atualizados) são retornados pela função.

A função de *backpropagation* apresentada no Quadro 7 é provavelmente a função mais importante para o aprendizado em redes neurais. Sua lógica consiste inicialmente em aplicar a função *feedforward* na rede (linhas 10 a 18) a fim de chegar a uma hipótese de resultado. A diferença aqui é que, ao invés de utilizar a função convencional, onde é retornado apenas o resultado referente a última camada, essa função mantém os estados de todas as camadas conforme aplica a função *feedforward*. Essa preservação dos estados é realizada para que a função seja então capaz de percorrer a rede no sentido oposto, partindo da última camada até a primeira (linhas 20 a 28). Conforme esse percurso inverso ocorre, é calculado a taxa de erro da rede através de uma função de custo, aplicada para cada camada (linhas 20 a 22 para a segunda camada e 24 a 28 para a primeira). Dessa forma, a função busca ajustar todos os pesos da rede de forma proporcional à sua contribuição para geração do erro (GOLDSCHMIDT, 2010, p. 91).

Além da função *sigmoid* utilizada no processo de *feedforward*, também é utilizado a função *sigmoid\_derivative* no processo de cálculo do erro, que consiste apenas na derivada da função *sigmoid*.

**Quadro 7. Retro propagação do erro**

1.	function [b2, b3, w1, w2] = backpropagation(x, y, bias2, bias3, weight1,
2.	weight2)
3.	
4.	# inicialização das variáveis w1, w2, b2 e b3
5.	# com as mesmas dimensões dos pesos e bias, porém, valendo zero
6.	
7.	activation = x;
8.	activations{1} = x;
9.	
10.	z = (weight1 * activation) + bias2;
11.	zs{1} = z;
12.	activation = sigmoid(z);
13.	activations{2} = activation;
14.	
15.	z = (weight2 * activation) + bias3;
16.	zs{2} = z;
17.	activation = sigmoid(z);
18.	activations{3} = activation;
19.	
20.	delta = (activations{3} - y) .* sigmoid_derivative(zs{2});
21.	b3 = delta;
22.	w2 = delta * activations{2}';
23.	
24.	z = zs{1};
25.	sp = sigmoid_derivative(z);
26.	delta = (weight2' * delta) .* sp;
27.	b2 = delta;
28.	w1 = delta * activations{1}';
29.	endfunction

Aqui destacam-se algumas características interessantes da linguagem Octave. Na linha 8 é possível observar a criação de uma lista de células (*cell array*), caracterizada por um rótulo de variável seguida de abertura e fechamento de chaves. Essa estrutura permite armazenar inúmeros valores e variáveis como em um vetor, porém com o diferencial de permitir que esses valores sejam de tipos e formatos diferentes (EATON, 2018). Dessa forma é possível utilizar essa estrutura para armazenar, como no código do Quadro 5, uma lista de matrizes e reutiliza-las depois.

Outro ponto é o uso do caractere aspas simples ( ' ) logo após uma matriz, utilizado nas linhas 22, 26 e 28. Utilizar esse caractere após uma matriz é equivalente a solicitar a matriz transposta da matriz original, onde as linhas são transformadas em colunas e as colunas são transformadas em linhas.

Por fim, na linha 26 são realizadas duas multiplicações, sendo que a primeira delas utiliza apenas o operador comum de multiplicação (\*) e a segunda utiliza o caractere ponto juntamente com o operador (. \*). Por ser uma linguagem desenvolvida especialmente para a manipulação de matrizes, Octave utiliza a multiplicação matricial como operação padrão de multiplicação, onde as linhas da primeira matriz são multiplicadas pelas colunas da segunda. Caso seja necessário realizar uma multiplicação simples, onde cada elemento da primeira matriz é multiplicado pelo elemento correspondente da segunda matriz, utiliza-se o caractere ponto junto com o operador de multiplicação (EATON, 2018).

A última função a ser apresentada é a *check\_input*, mostrada no Quadro 8, que recebe uma imagem contendo um número como entrada e retorna a hipótese alcançada pela rede para o número contido na imagem.

**Quadro 8. Validação da imagem submetida a rede**

1.	function answer = check_input (weight1, weight2, bias2, bias3, image_path)
2.	
3.	image = double(imread(image_path));
4.	image = sum(255 - image, 3);
5.	image = image / 255;
6.	input = reshape(image,1,[]);
7.	
8.	z = weight1 * input + bias2;
9.	output = sigmoid(z);
10.	z = weight2 * output + bias3;
11.	output = sigmoid(z);
12.	
13.	max = 0;
14.	answer = 1;
15.	for i = 1: size(output, 1)
16.	if (output(i) > max)
17.	max = output(i);
18.	answer = i;
19.	endif
20.	endfor
21.	answer -= 1;
22.	imagesc(reshape(input, 28, 28));
23.	printf("Hipotese: %d\n", answer);
24.	endfunction

A função recebe como parâmetro os pesos e bias com os valores já definidos após o treino da rede (retornados pela função *start\_network\_training*) e um último parâmetro contendo o endereço da imagem que será classificada. As linhas 3 a 6 leem essa imagem do disco como uma matriz, normalizam ela em valores entre 0 e 1, e a remodelam, de uma matriz de 28x28 para uma matriz de 784x1, de modo a ser compatível com a primeira camada da rede.

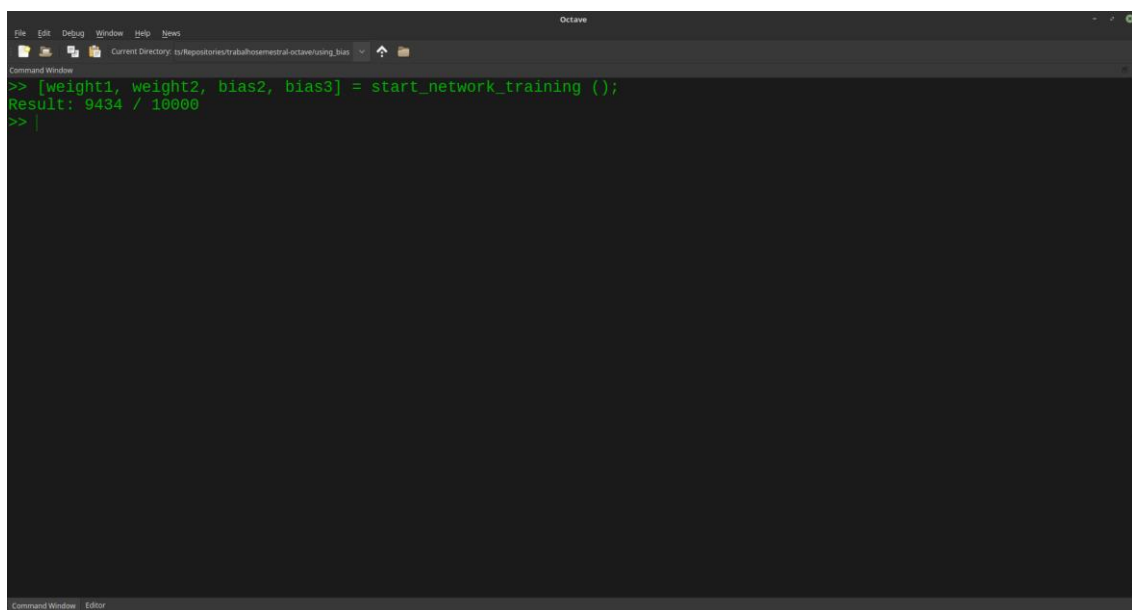
Em seguida, nas linhas 8 a 11, é aplicado a função *feedforward* em cada camada junto com a função *sigmoid*, e por fim, nas linhas 13 a 21, percorre-se todos os valores

da última camada buscando pelo valor mais alto contido nela. Como cada posição dessa camada representa um número entre 0 e 9, a posição que possuir o valor mais alto representa a hipótese alcançada pela rede do número mais compatível com o da imagem. A linha 22 apenas reconstrói e exibe a imagem original e a linha 23 exibe a hipótese alcançada pela rede.

## 2.5. Operacionalidade

O projeto não faz uso de interface gráfica, portanto os comandos de treino e validação das imagens são realizados via linha de comando na própria interface gráfica do Octave.

Antes de submeter uma imagem para ser validada é preciso primeiramente treinar a rede através da função *start\_network\_training*, como mostrado na Figura 2. Esse processo pode levar vários minutos, dependendo da configuração do equipamento utilizado. Como mencionado anteriormente, é possível diminuir a quantidade de *epochs* para agilizar o processo de treino, porém, isso pode resultar em uma diminuição da precisão da rede.

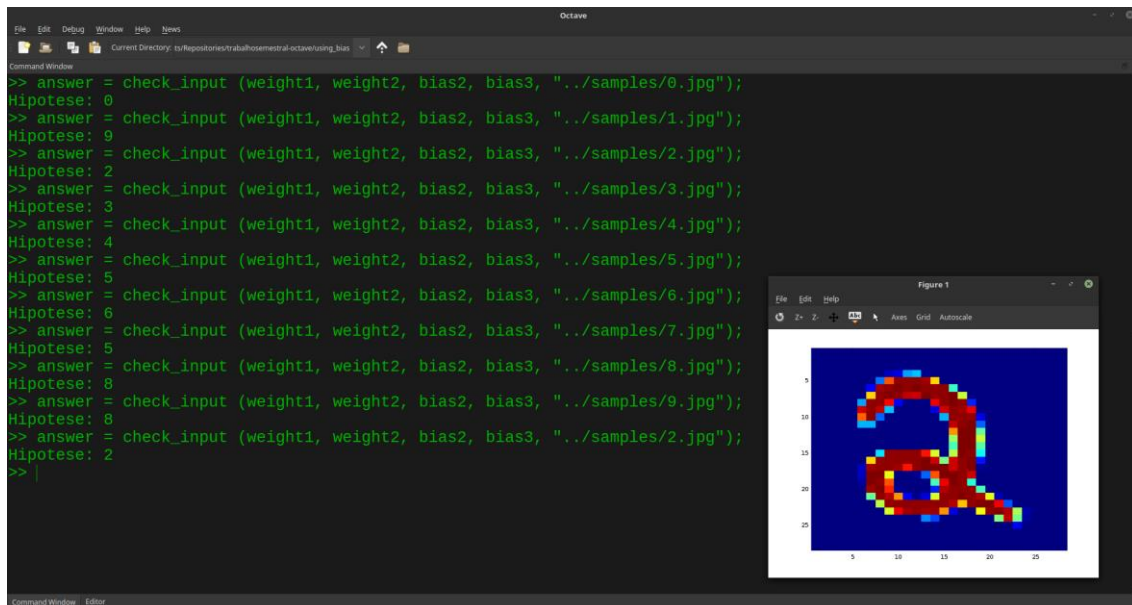
The image shows a screenshot of the Octave software interface. At the top, there is a menu bar with options like 'File', 'Edit', 'Debug', 'Window', 'Help', and 'Tools'. Below the menu bar, a status bar indicates the 'Current Directory' as 'c:\repositorios\trabalho\semestral-octave\using\_hiss'. The main area is a 'Command Window' with a dark background. It displays the following text: a prompt '>>' followed by the command '[weight1, weight2, bias2, bias3] = start\_network\_training();', the output 'Result: 9434 / 10000', and another prompt '>>' followed by a vertical bar '|'. The bottom of the window shows a 'Command Window Editor' tab.

**Figura 2. Treino da rede**

Quando o treino for finalizado uma mensagem como a mostrada na Figura 2 será exibida no terminal, indicando a quantidade de imagens que a rede classificou corretamente. Essas imagens utilizadas para testar a rede são as 10.000 imagens pertencentes a base de dados MNIST, reservadas especificamente para testes. Esse teste é executado pela função *validate\_hipoteses* que faz exatamente o mesmo processo utilizado na função *check\_input*, porém para as 10.000 imagens de teste.

Finalmente, após o treino, é possível chamar a função *check\_input* passando os pesos, bias e localização da imagem a ser validada como parâmetros, sendo que a imagem deve possuir dimensões de 28x28 pixels com fundo branco e o número escrito em cor preta. Ao executar a função, o resultado será exibido automaticamente, junto com uma nova janela exibindo a reconstrução da imagem fornecida como entrada. A Figura 3 mostra algumas tentativas realizadas, onde o nome do arquivo da imagem

passada por parâmetro corresponde ao número contido nela. Das dez imagens submetidas, sete foram classificadas corretamente e três incorretamente (sendo eles os números 1, 7 e 9).



**Figura 3. Classificação de algumas imagens pela rede**

#### 4. Discussões

Este artigo apresentou a implementação de uma rede neural artificial capaz de reconhecer e classificar dígitos manuscritos a partir de imagens, utilizando a linguagem de programação Octave e a base de dados MNIST.

A rede desenvolvida, ao utilizar as configurações de treino, arquitetura e imagens de treino apresentadas, atingiu uma precisão de, em média, 94%. Portanto é seguro dizer que os objetivos propostos inicialmente foram alcançados e a rede cumpre seu propósito de reconhecer dígitos manuscritos em imagens.

O Octave se mostrou uma linguagem de fácil compreensão, tendo uma boa legibilidade e facilidade de escrita. O fato da linguagem possuir poucas estruturas de dados e utilizar tipagem dinâmica em suas variáveis facilitou o aprendizado, de forma a não ser necessário um estudo prévio muito aprofundado da linguagem antes de utilizá-la. Outra vantagem foi sua semelhança com outras linguagens como C e PHP, o que facilitou a compreensão dos códigos.

Além da linguagem, também foi possível explorar e desmistificar um paradigma de programação diferente dos convencionais, que são as Redes Neurais Artificiais. Tal paradigma baseia-se em modelos matemáticos inspirados no funcionamento do cérebro humano e foi interessante compreender como essa arquitetura de camadas, neurônios e conexões sinápticas é traduzida para expressões matemáticas e implementadas em código através de uma linguagem de programação.

## **Referências**

Nielsen, M. (2017) “Neural Networks and Deep Learning”, <http://neuralnetworksanddeeplearning.com/index.html>, Junho.

GOLDSCHMIDT, R. R. Uma introdução à Inteligência Computacional: Fundamentos, Ferramentas e Aplicações. 1ª edição.

Octave (2018) “GNU Octave”, <https://www.gnu.org/software/octave/>

GUARDIA, L. E. (2003) “Redes Neurais com Funções de Ativação Adaptativas”.

LeCun Y., Cortes C., Burges C. J. C. (XXXX) THE MNIST DATABASE of handwritten digits, <http://yann.lecun.com/exdb/mnist/>, Junho.

John W. Eaton (2018) “GNU Octave”, <https://octave.org/doc/v4.2.2/>, Junho.