Implementação e Aplicação do Algoritmo Backpropagation para Classificação de Mamografias

Rodrigo Flexa*,
* Univesidade Federal do Pará (UFPA), Belém – Brazil
Email: rodrigo.flexa@itec.ufpa.br

***Resumo*—Este trabalho aborda o treinamento de redes neurais artificiais, mais especificamente a implementação e aplicação do algoritmo Backpropagation sem o uso de bibliotecas e ferramentas estabelecidas. Nesse sentido, a construção dessa rede neural de forma personalizada permite um estudo aprofundado do seu funcionamento interno além de flexibilidade na configuração da arquitetura da rede. Portanto, faremos a construção dessa rede neural e em seguida avaliaremos o seu desempenho a um problema de classificação utilizando dados de mamografias.**

***Palavras-Chave*—BackPropagation, Redes Neurais, Descida do Gradiente Estocástico.**

I. Introdução

Redes Neurais estão cada vez mais em alta. A pesquisa envolvendo esse tema cresceu exponencialmente nos últimos anos, despertando a curiosidade das pessoas no tema e nas aplicações que podem surgir com o estudo dessa tecnologia. No entanto, o funcionamento matemático dessa tecnologia frequentemente é abstraído por bibliotecas e frameworks amplamente utilizados, como TensorFlow e PyTorch, ocultando detalhes essenciais de seu comportamento. Nesse sentido, o principal intuito desse trabalho é descrever uma implementação de rede neural do ”zero”. Para isso, faremos uso da linguagem de programação *python* além de algumas bibliotecas que facilitam cálculos matemáticos, como o *numpy*.

A rede neural construída será simplificada para uma camada escondida (com número variável de neurônios nessa camada) e fará uso da *Descida do Gradiente Estocástico (SGD)*. Ou seja, para cada exemplo de treino, faremos a atualização dos pesos e biases da nossa rede. A rede permitirá também a troca da função de ativação conforme desejado. Nesse trabalho incluímos as seguinte funções de ativação: *Linear, Sigmoid, Tangente Hiperbólica e ReLu*. Para melhor visualização das atualizações, será possível verificar os valores dos pesos sinápticos antes e após a finalização do treinamento.

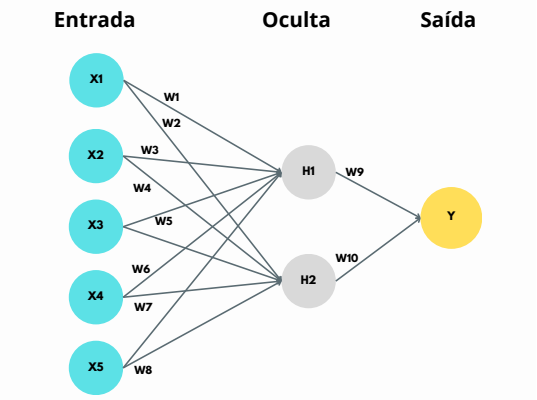
Após a construção da rede neural, usaremos ela para classificar dados de mamografia que compreendem cinco variáveis de entrada: Avaliação BIRADS, idade, forma, margem e densidade. A saída desejada é uma classe binária, sendo 0 para mamografias consideradas ”normais”e 1 para aquelas classificadas como ”anormais”. Testaremos diversas topologias e usaremos como métrica para determinar a melhor topologia da rede neural a acurácia com base em um procedimento de busca em grade. Além disso, levaremos em conta uma análise envolvendo Falsos Negativos, ou seja, a mamografia era anormal, porém o modelo a classificou como normal. Este tipo de erro é crítico em diagnósticos médicos, pois a detecção precoce de anormalidades pode ser vital para o tratamento.

II. Fundamentação Teórica

As Redes Neurais Artificiais (RNAs) são modelos computacionais inspirados no funcionamento do cérebro humano, que consistem em neurônios artificiais conectados em camadas, tendo cada camada uma quantidade configurável de neurônios. O funcionamento desse neurônio artificial ocorre da seguinte maneira: recebe um conjunto de entradas x_i ponderadas por pesos w_i associados, realiza a soma dessas entradas com um viés b, aplica uma função de ativação f, resultando na saída y desse neurônio. Cada camada de uma rede neural possui uma quantidade de neurônios.

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

Quando temos uma ou mais camadas intermediárias na rede, ela é chamada de Perceptron Multicamadas (MLP - Multi-Layer Perceptron). No caso da MLP que iremos desenvolver, teremos uma camada de entrada com 5 neurônios (um para cada feature), uma camada de saída com um único neurônio, e uma camada escondida com um número configurável de neurônios. Como nossa rede terá apenas uma camada escondida, ela será o que é conhecido como *rede neural rasa*. Se considerarmos 2 neurônios na camada escondida, a rede ficará semelhante à figura abaixo (omitindo os viéses):



Nesse cenário hipotético os pesos entre a entrada e a camada oculta (W_H) e os biases (b_H) são Além d

$$W_H = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \\ w_5 & w_6 \\ w_7 & w_8 \\ w_9 & w_{10} \end{pmatrix}, \quad b_H = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

Além disso, os pesos entre a camada oculta e a camada de saída (W_O) e o bias da saída (b_O) são definidos como:

$$W_O = \begin{pmatrix} w_9 \\ w_{10} \end{pmatrix}, \quad b_O = b_Y$$

As ativações da camada oculta N e da saída $\hat{Y}$ são dadas por:

$$N = f(W_H X + b_H)$$
$$\hat{Y} = f(W_O^T N + b_O)$$

Quando aplicamos um determinado vetor de entrada dos nossos dados de treino e calculamos a saída $\hat{Y}$, estamos realizando o que chamamos de *forward*. Num exemplo de classificação, como o nosso, $\hat{Y}$ representa a previsão da rede com base nas entradas X_i. Com o vetor de entrada X_i calculamos N e a partir dele calculamos a saída $\hat{Y}$.

Para que a rede neural aprenda a fazer previsões mais precisas, utilizamos o algoritmo de *backpropagation*, que ajusta os pesos e biases com base no erro das previsões. O erro é calculado utilizando uma função de perda L, que mede a discrepância entre o valor real Y e a previsão $\hat{Y}$ da rede. Nesse caso, utilizamos o Erro Médio Quadrático (MSE), definido para esse caso específico como:

$$L = \frac{1}{2}(Y - \hat{Y})^2$$

O objetivo do *backpropagation* é minimizar a função de perda ajustando os pesos e biases da rede neural. Para isso, o erro calculado na camada de saída é propagado para trás na rede, e são calculados os gradientes da função de perda L em relação aos pesos W e biases b.

O primeiro passo é calcular o gradiente da função de perda em relação à previsão $\hat{Y}$:

$$\delta_O = \hat{Y} - Y$$

Agora, o gradiente em relação aos pesos da camada de saída W_O é calculado como:

$$\nabla_{W_O} L = \delta_O \cdot N^T$$

E em relação aos biases da camada de saída:

$$\nabla_{b_O} L = \delta_O$$

De forma semelhante, o gradiente dos pesos da camada oculta W_H é calculado propagando o erro de volta a partir da camada de saída. O erro propagado para a camada oculta é dado por:

$$\delta_H = (W_O^T \cdot \delta_O) \cdot f'(z_H)$$

Aqui: - W_O^T é a transposta dos pesos da camada de saída. - δ_O é o erro na camada de saída. - $f'(z_H)$ é a derivada da função de ativação da camada oculta em relação a $z_H = W_H X + b_H$.

Agora podemos calcular o gradiente em relação aos pesos W_H da camada oculta:

$$\nabla_{W_H} L = \delta_H \cdot X^T$$

E em relação aos biases da camada oculta:

$$\nabla_{b_H} L = \delta_H$$

Após o cálculo dos gradientes, os pesos e biases são atualizados utilizando o algoritmo de Descida do Gradiente Estocástico (SGD). A atualização é dada por:

$$W_i^{\text{novo}} = W_i - \eta \nabla_{W_i} L, \quad b_i^{\text{novo}} = b_i - \eta \nabla_{b_i} L$$

Onde: - η é a taxa de aprendizado, que controla o tamanho do passo na direção do gradiente. - $\nabla_{W_i} L$ e $\nabla_{b_i} L$ são os gradientes da função de perda em relação aos pesos W_i e biases b_i, respectivamente.

III. Metodologia

Nesta seção, descrevemos o processo adotado para o desenvolvimento e avaliação do modelo de rede neural utilizado na classificação de dados de mamografia.

A. Organização do código

A rede neural deverá atender a determinados requisitos do projeto, como: a possibilidade de verificar os pesos e biases da rede, a configuração da quantidade de neurônios na camada escondida e funções de ativação, validação cruzada e parada antecipada (60% dos dados para treino, 20% para validação e 20% para teste). A parada antecipada ocorrerá caso o erro médio quadrático (MSE) na base de validação aumente após 5 épocas consecutivas.

Para atender aos requisitos do projeto, a implementação do algoritmo de *Backpropagation* foi modularizada em dois arquivos Python: *funcoes_ativacao.py*, responsável por conter as funções de ativação e suas derivadas, e o arquivo *nn.py*, que contém o código da rede neural em si, sendo assim, a quantidade de neurônios em cada camada, pesos, bias, como também as funções de forward, backpropagation e de treinamento. Além disso, foi criada uma pasta *utils* contendo os arquivos *normalizador.py* e *spliter.py*, os quais são úteis para o tratamento e a preparação dos dados.

Por fim, para rodar e avaliar a rede neural no nosso conjunto de dados de mamografia, utilizamos um notebook Jupyter chamado *analysis.ipynb*.[1]

[1]O código completo pode ser acessado em: https://github.com/RodrigoFlexa/Backpropagation

B. Pré-processamento dos Dados

O dataset de mamografias é inicialmente lido em um *DataFrame* do `pandas`, e, em seguida, suas colunas foram renomeadas.

```
dataset= pd.read_excel("dados.xlsx")
cols=['x1','x2','x3','x4','x5','y']
dataset.columns= cols
```

Para normalização os dados de entrada, utilizamos a técnica de MinMaxScaler na classe `Normalizador`, localizada no arquivo `normalizador.py`. A normalização dos dados é realizada para garantir que todas as variáveis de entrada fiquem dentro do intervalo [0, 1], visto que o modelo pode ser afetado pela discrepaância entre os dados. A fórmula do *MinMaxScaler* é dada por:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

onde:

- X é o valor original da característica a ser normalizada,
- X_{min} é o valor mínimo da característica no conjunto de dados,
- X_{max} é o valor máximo da característica no conjunto de dados,
- X_{norm} é o valor da característica normalizada.

Além disso, o conjunto de dados foi dividido em três subconjuntos: treino (60%), validação (20%) e teste (20%). A separação foi realizada utilizando a classe `Splitter`, localizada no arquivo `spliter.py`. Essa divisão garante que o modelo seja treinado em um conjunto de dados e avaliado em dados não vistos anteriormente durante o processo de validação e teste.

```
splitter = Splitter(dataset_normalizado)
treino, teste, validacao =
splitter.split_data(0.6, 0.2, 0.2)
```

C. Estrutura da Rede Neural

A rede neural implementada possui uma arquitetura com uma camada de entrada, uma camada escondida e uma camada de saída. A quantidade de neurônios na camada escondida é configurável, permitindo flexibilidade para ajustar a rede de acordo com a complexidade dos dados. A classe `NeuralNetwork` é inicializada com os seguintes parâmetros:

- `n_entradas`: Número de entradas da rede.
- `n_saidas`: Número de saídas da rede.
- `n_neuronios_escondidos`: Número de neurônios na camada escondida.
- `func_ativacao`: Função de ativação utilizada. Pode ser `sigmoid`, `relu`, `linear` ou `tanh`.
- `seed`: Parâmetro opcional para inicialização da semente de geração de números aleatórios.

Os pesos e biases são inicializados aleatoriamente, e o histórico de erro e acurácia é armazenado em listas para posterior análise.

A rede neural foi inicializada com pesos aleatórios seguindo uma distribuição normal, e os biases foram inicializados como zeros. O treinamento foi realizado utilizando o algoritmo de *backpropagation*, que ajusta os pesos e biases da rede com base nos gradientes calculados a partir da função de perda.

D. Treinamento da Rede Neural

A função principal para o ajuste dos parâmetros da rede neural durante o treinamento é a função `treinar`. Ela realiza várias iterações (épocas) nas quais os pesos e *biases* da rede são atualizados com o objetivo de minimizar o erro no conjunto de treino. O método inclui a validação cruzada para monitorar o desempenho do modelo em um conjunto de validação, além da possibilidade de parar o treinamento de forma antecipada se o erro de validação não melhorar após um número definido de épocas. Porém, antes de falar dela precisamos descrever um pouco melhor as funções de `foward` e de `backpropagation`.

A função `forward` é responsável por calcular a saída da rede neural com base nas entradas fornecidas. Ela realiza a propagação da informação da camada de entrada até a camada de saída, passando pela(s) camada(s) escondida(s).

Função `forward`

Função `forward`

Parâmetros:

- **X**: Matriz de entrada com os dados de entrada para a rede (dimensão $n\_samples \times n\_entradas$).

Retorno:

- A saída da rede neural após a aplicação das funções de ativação.

Descrição: A função `feedforward` realiza os seguintes passos:

1) A matriz de entrada **X** é multiplicada pelos pesos da camada de entrada, somada ao *bias* correspondente, resultando na entrada da camada escondida.
2) A função de ativação escolhida (sigmoid, *ReLU*, *tanh*, *linear*) é aplicada à entrada da camada escondida, resultando na sua saída.
3) A saída da camada escondida é multiplicada pelos pesos da camada de saída, somada ao *bias* correspondente, resultando na entrada da camada de saída.
4) Finalmente, a função *sigmoid* é aplicada à entrada da camada de saída para produzir a predição final da rede.

Essa predição, resultante da passagem dos dados pelas camadas da rede e pelas funções de ativação, representa a

tentativa da rede em estimar o valor esperado com base nos padrões aprendidos.

Essa predição é então utilizada pela função `backpropagation`, que é executada logo após a função `forward`, pois depende das predições geradas para calcular o erro e propagar essa informação de volta pelas camadas da rede. Esse ciclo contínuo de `forward` e `backpropagation` é o que permite que a rede aprenda com os dados. Abaixo está uma descrição de como a função de `backpropagation` foi implementada.

Função `backpropagation`

Parâmetros:

- **X**: Matriz de entrada com os dados de treino (dimensão $1 \times n\_entradas$).
- **y**: Vetor com a saída esperada para os dados de treino (dimensão $1 \times n\_saidas$).
- **taxa_aprendizagem**: A taxa de aprendizado que controla o tamanho do ajuste dos pesos (um valor pequeno, geralmente na ordem de 0.01).

Retorno:

- A função não retorna valores diretamente, mas ajusta os pesos e *biases* da rede de acordo com o erro calculado.

Descrição: A função `backpropagation` segue os seguintes passos:

1) A função `feedforward` é chamada para calcular a saída da rede com base nas entradas fornecidas.
2) O erro é calculado como a diferença entre a saída prevista pela rede e a saída real **y**.
3) cálculo dos deltas: O delta da camada de saída é calculado multiplicando o erro pela derivada da função de ativação da camada de saída. O delta da camada escondida é calculado propagando o erro de volta da camada de saída para a camada escondida, ajustado pela derivada da função de ativação.
4) Atualização dos pesos e *biases*: Os pesos e *biases* são atualizados de acordo com os deltas calculados e a taxa de aprendizado.

Finalmente, a função `treinar` é o componente central do processo de aprendizado. Ela executa o ciclo de treinamento, ajustando os pesos da rede a cada época e monitorando o desempenho no conjunto de validação, visto a parada antecipada. Nesse sentido, a função de treinar irá a cada época passar os dados de treino pela rede usando a função `forward` e atualizá-la através do uso `backpropagation`. No fim da época irá avaliar o erro de validação.

Função `treinar`

Parâmetros:

- **X**: Matriz de entrada com os dados de treino.
- **y**: Vetor com as saídas esperadas para os dados de treino.
- **X_validacao**: Matriz de entrada com os dados de validação.
- **y_validacao**: Vetor com as saídas esperadas para os dados de validação.
- **epochs**: Número máximo de épocas de treinamento (padrão: 1000).
- **taxa_aprendizagem**: Taxa de aprendizado para ajuste dos pesos (padrão: 0.01).
- **early_stopping_limit**: Número de épocas consecutivas sem melhora no erro de validação, após o qual o treinamento é interrompido (padrão: 5).

Retorno:

- Não há retorno explícito, mas os pesos e *biases* da rede são ajustados ao longo do treinamento.

Descrição:

1) A cada época, os dados de entrada **X** são passados pela rede através da função de *feedforward*, que calcula a saída prevista pela rede.
2) Em seguida, o erro é calculado com base na diferença entre a saída prevista e a saída real **y**. Esse erro é utilizado no processo de *backpropagation* para ajustar os pesos e *biases* da rede.
3) O erro MSE é calculado para o conjunto de validação.
4) O treinamento continua até atingir o número máximo de épocas ou até que a condição de parada antecipada (*early stopping*) seja ativada.

A cada época, o erro de treino e de validação é registrado, junto com a acurácia de validação, o que permite ao usuário monitorar o desempenho do modelo ao longo do treinamento.

Além das funções principais de treinamento, a classe `NeuralNetwork` conta com funções adicionais que facilitam o processo de avaliação e inspeção da rede, como a função `prever` e `mostrar_pesos`.

Função `mostrar_pesos`

Parâmetros:

- **my_string**: Identifica o conjunto de pesos.

Retorno:

- A função não retorna valores, mas imprime os pesos e *biases* de cada camada da rede.

Descrição: A função `mostrar_pesos` permite a visualização dos pesos e *biases* atuais da rede. A string fornecida em **my_string** é usada como um identificador para facilitar a interpretação da impressão dos pesos.

Para que seja feito o monitoramento de erros e acurácia usaremos a função `calcular_mse`, a qual é utilizada para medir o erro da rede com base na função *Mean Squared Error* (MSE) e a função `calcular_acuracia` para avaliar a taxa de acerto das previsões.

Função `calcular_mse`

Parâmetros:

- **X**: Matriz de entrada com os dados para os quais se deseja calcular o erro.
- **y**: Vetor com as saídas reais esperadas.

Retorno:

- O valor do erro MSE calculado.

Descrição: A função realiza o cálculo do erro MSE utilizando a fórmula:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2$$

onde y_i são os valores reais e $\hat{y_i}$ são os valores previstos pela rede.

Função `calcular_acuracia`

Parâmetros:

- **X**: Matriz de entrada com os dados para os quais se deseja calcular a acurácia.
- **y**: Vetor com as saídas reais esperadas.

Retorno:

- A acurácia da rede, medida como a proporção de previsões corretas em relação ao total.

Descrição: A função compara as previsões geradas pela rede com as saídas reais e calcula a taxa de acerto, retornando um valor entre 0 e 1.

E. Avaliação da Rede Neural

A avaliação da rede neural foi conduzida utilizando múltiplos conjuntos de dados (treino, validação e teste) e diferentes configurações de hiperparâmetros para conseguir generalizar melhor o modelo. O método utilizado para essa avaliação foi a técnica de busca em grade (*grid search*), na qual diferentes quantidades de neurônios e funções de ativação foram combinadas para treinar a rede, permitindo a identificação das configurações mais eficientes.

Foram testadas múltiplas combinações de números de neurônios nas camadas escondidas e funções de ativação. Assim, foram testadas seguintes configurações de neurônios na camada escondida: [16,32,64]. Quanto as funções de ativação foram testadas as opções como `sigmoid`, `relu`, `tanh` e `linear`. Esse processo permitiu avaliar o impacto de cada uma dessas variáveis no desempenho final da rede.

Para cada combinação de neurônios e funções de ativação, o modelo foi treinado com o conjunto de dados de treino (`X_treino` e `y_treino`) por até 1000 épocas. Durante o treinamento, o desempenho da rede foi monitorado utilizando o conjunto de validação (`X_validacao` e `y_validacao`), com o objetivo de medir o *Mean Squared Error* (MSE) e a acurácia. Após o término do treinamento, o conjunto de teste (`X_teste` e `y_teste`) foi utilizado para avaliar a capacidade do modelo de generalizar para dados novos, também medindo o MSE e a acurácia nesse conjunto.

As principais métricas usadas para avaliar o desempenho do modelo em diferentes configurações foram:

- **MSE (Erro Médio Quadrático)**: Uma métrica de erro que mede a diferença média entre os valores reais e as previsões da rede. Quanto menor o MSE, melhor a rede está ajustada aos dados.
- **Acurácia**: A proporção de predições corretas feitas pela rede em relação ao total de exemplos testados. Essa métrica foi utilizada tanto no conjunto de validação quanto no conjunto de teste.

A busca em grade também revelou o impacto do número de neurônios e da função de ativação sobre o tempo de treinamento. Redes com mais neurônios tendiam a levar mais tempo para convergir, enquanto funções de ativação como `relu` apresentaram melhores resultados em termos de tempo e precisão.

IV. Resultados e Discussão

Após aplicarmos o processo grid search, iremos coletar os dados e buscar a configuração que nos retorna a maior acurácia de validação. A Tabela I apresenta os resultados obtidos para cada combinação de hiperparâmetros, destacando o número de neurônios, a função de ativação e a acurácia no conjunto de validação.

TABLE I
Resultados da busca por hiperparâmetros (Acurácia de Validação)

Neurônios	Função de Ativação	Acurácia de Validação
16	sigmoid	0.797619
16	relu	0.809524
16	linear	0.803571
16	tanh	0.797619
32	sigmoid	0.803571
32	relu	0.773810
32	linear	0.809524
32	tanh	0.797619
64	sigmoid	0.803571
64	relu	0.803571
64	linear	0.809524
64	tanh	0.791667

Com base nesses resultados, o modelo que apresentou a maior acurácia de validação foi o que utilizou 16 neurônios na camada escondida, com a função de ativação *relu*, e uma taxa de aprendizado de 0.01, obtendo uma acurácia de 0.8095 no conjunto de validação.

Avaliando melhor esse modelo escolhido, a Figura 1 apresenta a evolução do erro médio quadrático (MSE) ao longo das épocas. Nesse gráfico, observamos o comportamento tanto no conjunto de treino quanto no conjunto de validação.

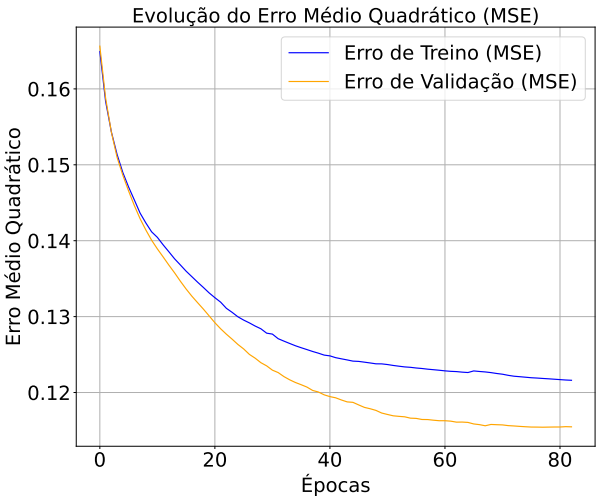


Fig. 1. Evolução do Erro Médio Quadrático (MSE) para o modelo com 16 neurônios e função de ativação *relu*.

A curva de erro de treino (azul) apresenta uma rápida diminuição nas primeiras épocas, estabilizando-se após aproximadamente 50 épocas, enquanto o erro de validação (laranja) também diminui de forma semelhante. Esse comportamento sugere que o modelo está aprendendo bem, reduzindo o erro tanto no conjunto de treino quanto no de validação. Após 50 épocas, o erro de validação começa a estabilizar e até a diminuir, o que pode indicar uma boa capacidade de generalização do modelo, sem sinais claros de overfitting.

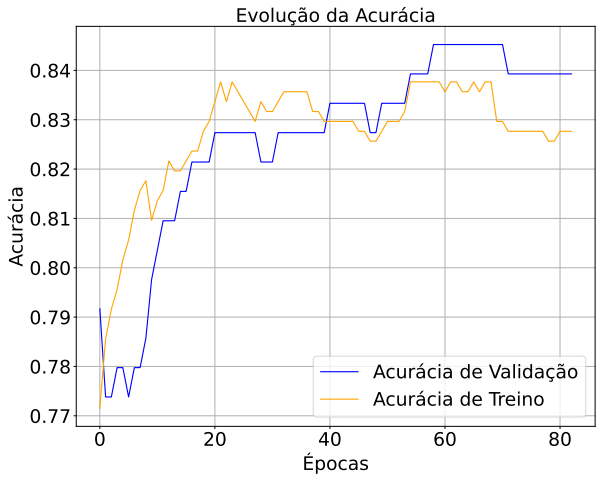


Fig. 2. Evolução da Acurácia para o modelo com 16 neurônios e função de ativação *relu*.

A Figura 2 mostra a evolução da acurácia ao longo das épocas. A acurácia no conjunto de validação (linha azul) sobe rapidamente nas primeiras 20 épocas, alcançando uma estabilização próxima de 82% a 83% após 50 épocas. A acurácia de treino (linha laranja) também aumenta rapidamente, estabilizando-se em torno de 83%. A proximidade entre as curvas de treino e validação é um bom sinal de que o modelo está se ajustando bem aos dados de treino e generalizando corretamente para o conjunto de validação, sem indícios evidentes de overfitting.

Além da análise do erro médio quadrático (MSE) e da acurácia, também é importante verificar a matriz de confusão, que nos fornece uma visão detalhada dos acertos e erros do modelo nas classes analisadas. A Figura 3 apresenta a matriz de confusão gerada a partir das previsões no conjunto de teste, permitindo avaliar a capacidade do modelo em identificar corretamente as classes.

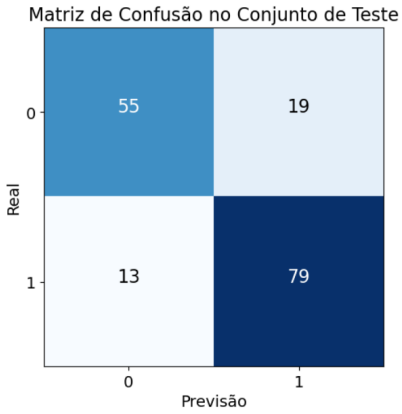


Fig. 3. Matriz de confusão no conjunto de teste.

Analisando a matriz, podemos observar os seguintes pontos:

- O modelo classificou corretamente 55 instâncias da classe 0 e 79 instâncias da classe 1.
- Foram identificados 19 falsos positivos (previsões incorretas de classe 1, quando a classe real era 0).
- Foram encontrados 13 falsos negativos (previsões incorretas de classe 0, quando a classe real era 1).

Os falsos negativos são particularmente preocupantes em problemas de classificação relacionados a diagnósticos médicos, pois representam casos em que o modelo falhou em identificar pacientes com dados alterados, classificando-os incorretamente como normais. No total, 13 casos de falsos negativos indicam que o modelo não detectou corretamente aproximadamente 14% dos casos reais da classe 1.

Além disso, os 19 falsos positivos sugerem que o modelo identificou incorretamente 26% dos casos da classe 0 como pertencentes à classe 1.

Para avaliar o desempenho global, podemos calcular as seguintes métricas:

- **Precisão da Classe 1**: $\frac{79}{79+19} = 80.61\%$
- **Recall da Classe 1)**: $\frac{79}{79+13} = 85.87\%$
- **F1-Score da Classe 1**: $\frac{2 \cdot 80.61 \cdot 85.87}{80.61+85.87} \approx 83.16\%$

Com base nesses resultados, podemos concluir que o modelo tem um bom desempenho, especialmente em termos de *recall* para a classe 1, o que indica que a maioria dos casos positivos foi corretamente identificada. No entanto, o número

de falsos positivos e falsos negativos indica que há espaço para melhorias, especialmente no que diz respeito à minimização dos falsos negativos, que são críticos em diagnósticos médicos.

A redução do número de falsos negativos poderia ser alcançada com ajustes no limiar de decisão do modelo ou através de técnicas de balanceamento de classes, como o aumento da penalização para erros na classe 1.

V. Conclusão

Implementamos uma rede neural utilizando *Backpropagation* para classificar dados de mamografia, explorando diferentes configurações de hiperparâmetros. O melhor modelo, com 16 neurônios e função *relu*, alcançou 80.95% de acurácia. A análise da matriz de confusão mostrou a necessidade de reduzir falsos negativos, críticos em diagnósticos médicos. Ajustes no limiar de decisão e técnicas de regularização são sugeridos como trabalhos futuros para melhorar o desempenho e a generalização do modelo.