Relatório do Problema 4: 0 ou 1?

Inteligência Artificial

Licenciatura em Engenharia Informática

Faculdade De Ciências e Tecnologias - Universidade Do Algarve

Versão 1.0

Elaborado por:

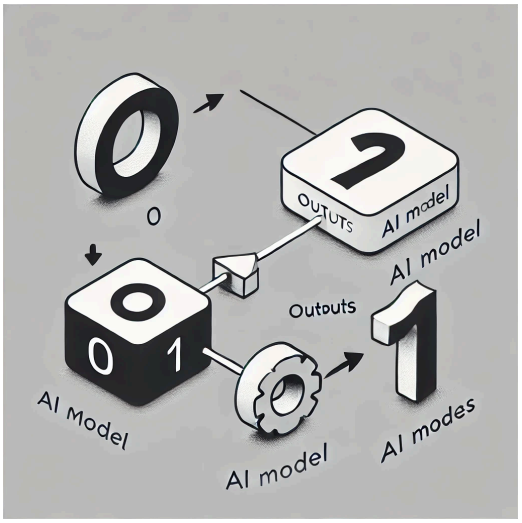
Pedro Ferreira Nº79389

Luís Rosa Nº81009

José Lima Nº79738

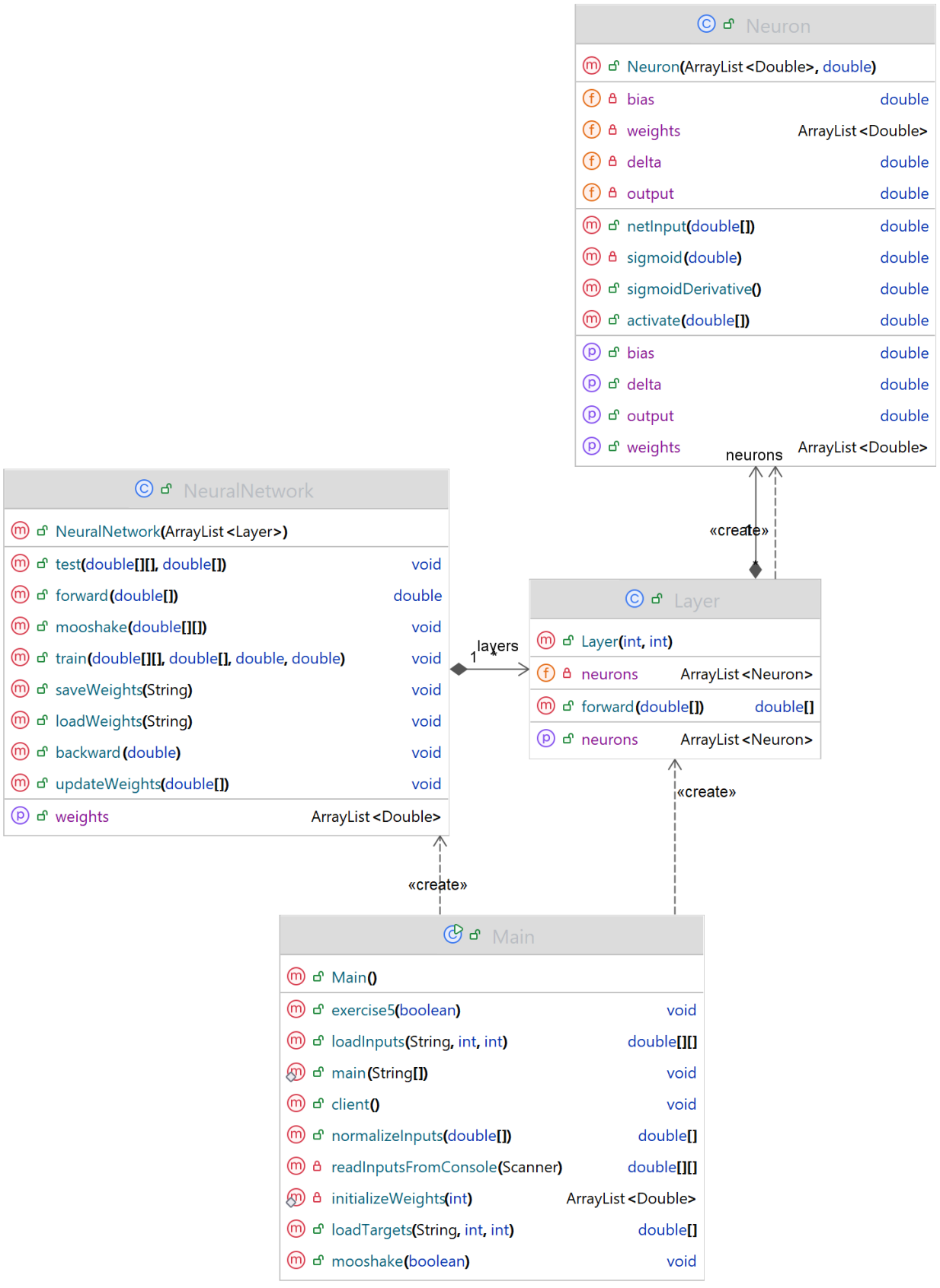
Grupo 1, PL3

09/12/2024



1. Índice

2. Diagrama de Classes UML (Gerado no IntelliJ)



3. Descrição do Problema e Algoritmos Utilizados

Descrição do Problema e Algoritmos Utilizados

O problema abordado neste trabalho consiste na classificação de dígitos manuscritos 0 e 1 a partir de imagens representadas por valores em “grayscale”, cada pixel da imagem representado por um valor numérico. Para tal, utilizamos um subconjunto simplificado do conjunto de dados MNIST, amplamente reconhecido pela sua eficácia no reconhecimento de números manuscritos. Cada imagem tem uma resolução de 20x20 píxeis, totalizando 400 valores que representam os níveis de intensidade do cinzento de cada pixel.

O principal objetivo deste trabalho foi treinar uma rede neuronal capaz de distinguir com precisão entre os dígitos 0 e 1, ao utilizar uma lista de dados fornecidos. O conjunto de dados disponibilizado contém 800 amostras, cada uma composta por 400 valores correspondentes aos píxeis de uma imagem.

Para garantir que o modelo treinado seja robusto e generalize bem para novos dados, adotámos algumas estratégias durante o processo de treino. Inicialmente, os dados de entrada foram normalizados para o intervalo [0, 1], o que contribui para a estabilidade e eficiência do treino da rede neuronal. Além disso, implementámos a técnica de validação cruzada (cross-validation), que nos permite avaliar o desempenho do modelo de forma mais fiável e diminuir o risco de overfitting. Esta abordagem assegura que o modelo não se limite a memorizar os dados de treino, mas que também capture padrões relevantes para a classificação correta de novos exemplos.

A técnica de validação cruzada utilizada foi a validação cruzada k-fold, onde o conjunto de dados é dividido em **k subconjuntos (folds)**, sendo cada um utilizado alternadamente como conjunto de teste enquanto os restantes servem como conjunto de treino. Neste caso, utilizámos um valor de k=3, permitindo-nos avaliar o desempenho do modelo em diferentes partições do conjunto de dados e obter uma estimativa fiável do erro médio de generalização. Testámos diferentes configurações de redes neuronais, variando o número de camadas e neurónios em cada camada. Graças à validação cruzada, conseguimos comparar os erros associados a cada configuração e, com base nos resultados, selecionar a arquitetura que apresentou melhor equilíbrio entre precisão e simplicidade. Esta abordagem, apesar de simples e não muito técnica, conseguiu de forma muito expressiva demonstrar-nos qual o modelo mais adequado para a tarefa, garantindo bom desempenho sem sobreajuste (overfitting) sem a necessidade de explorar dados técnicos do problema.

Além disso, a utilização de validação cruzada forneceu métricas como a precisão média e o RMSE entre os folds, o que nos ajudou a compreender a robustez das diferentes arquiteturas testadas. Desta forma, o processo de seleção do modelo tornou-se mais fundamentado e menos suscetível a variações nos dados.

Arquitetura da Rede Neuronal

- **Camada de Entrada**: A rede possui 400 entradas correspondentes aos 400 píxeis das imagens (valores greyscale).
- **Camadas Ocultas**: O número de camadas ocultas e de neurónios foi ajustado de forma experimental para encontrar o melhor equilíbrio entre desempenho e complexidade/simplicidade.
- **Camada de Saída**: Consiste num único neurónio com uma função de ativação sigmóide, retornando valores no intervalo [0, 1], que são utilizados para classificar as imagens como 0 ou 1 (arredondando o valor final).
- **Função de Ativação**: Função sigmóide em todos os neurónios.
- **Algoritmo de Treino**: O treino foi realizado utilizando retropropagação do erro (*backpropagation*) com descida do gradiente.

Processo de Treino

1. **Inicialização**: Os pesos da rede foram inicializados com valores aleatórios que foram reduzidos proporcionalmente, multiplicados por 0.1, de modo a suscitar uma mais fácil e rápida convergência dos mesmos.
2. **Normalização**: Os valores dos píxeis foram normalizados para o intervalo [0, 1] de forma proporcional, de modo a não existir perda de informação.
3. **Validação Cruzada (*Cross-Validation*)**:
    - O conjunto de dados foi dividido em *k* subconjuntos (neste caso, utilizou-se aproximadamente k=3).
    - Em cada iteração, um dos subconjuntos foi reservado como conjunto de teste, enquanto os restantes foram usados para treino.
    - O desempenho do modelo foi avaliado em cada subconjunto, e o erro médio foi registado.
4. **Retropropagação**:
    - O erro foi calculado para cada amostra com base na diferença entre a saída prevista e o valor esperado.
    - As derivadas das funções de ativação (sigmoide) foram utilizadas para ajustar os pesos.
5. **Paragem Antecipada**: O treino foi interrompido caso o erro médio quadrático (MSE) atingisse um limiar pré-definido (principal parâmetro de paragem) ou demonstrasse aumento após 10 épocas consecutivas (evitando casos de overfitting).
6. **Armazenamento dos Pesos**: Após o treino, os pesos da rede foram guardados para reutilização durante a inferência num ficheiro com o formato *csv*.

7. **Conjunto de Testes:** Após o treino utilizando validação cruzada, o modelo final foi avaliado no conjunto de teste (separado anteriormente do conjunto principal). As previsões foram comparadas com os targets reais para calcular a precisão (*accuracy*).

4. Padrões de Projeto

Neste projeto, várias decisões de design e padrões de projeto foram seguidos para garantir uma implementação eficiente, modular e fácil de manter do modelo de rede neuronal e dos respetivos processos de treino e validação.

1. Modulação Baseada em Componentes

O desenvolvimento foi estruturado com base nos seguintes módulos principais, cada um com responsabilidades bem definidas:

- **Pré-processamento de Dados**: Inclui normalização e separação dos conjuntos de treino, validação e teste.
- **Modelo de Rede Neuronal**: Implementação da arquitetura, que inclui inicialização, definição das camadas e funções de ativação.
- **Treino e Validação**: Contém os métodos para retropropagação e cálculo dos erros, incluindo validação cruzada.
- **Inferência**: Gere as previsões e o cálculo da precisão no conjunto de teste.

Este padrão de modularidade assegura que alterações num módulo (por exemplo, normalização dos dados) não afetam diretamente os restantes.

2. Padrão *Strategy* para Escolha de Hiperparâmetros

A seleção dos hiperparâmetros (como número de camadas, neurónios, taxa de aprendizagem e regularização) seguiu o padrão *Strategy*, permitindo testar diferentes configurações de forma sistemática:

- Foram criadas estratégias configuráveis, como diferentes taxas de aprendizagem.
- Cada estratégia foi avaliada usando validação cruzada, permitindo escolher a melhor abordagem com base em métricas objetivas.

3. Padrão *Observer* para Monitorização de Treino

Durante o treino, foi implementado um mecanismo de monitorização baseado no padrão *Observer*:

- O erro (mse) e a precisão foram monitorizados após cada época.

4. Padrão de Validação Cruzada (Metodologia)

A abordagem de validação cruzada implementou as melhores práticas para dividir o conjunto de dados:

- Divisão em *k* subconjuntos.
- Execução iterativa do treino e validação em subconjuntos distintos.
- Consolidação dos resultados para calcular métricas médias de desempenho.

Este método assegura uma avaliação robusta do modelo, mitigando o impacto de divisões acidentais dos dados.

5. Descrição das Classes

Classe Main

- **Responsável por orquestrar a execução do programa.**
  - Contém o método principal **`main()`** que inicia o programa.
  - Possui métodos como:
    - **`exercise5(boolean shouldTrain)`**: para treinar ou testar a rede neural, de acordo o argumento recebido.
    - **`client()`**: interface para escolher entre treinar ou testar a rede com valores anteriormente guardados, reutiliza o método ***exercise5***.
    - **`mooshake(boolean shouldTrain)`**: método alternativo para testar a rede, permitindo entrada manual de dados assim como é solicitada na plataforma de validação do código “mooshake”.
  - Métodos auxiliares:
    - **`loadInputs()`** e **`loadTargets()`**: carregam dados de entrada e saída de arquivos CSV.
    - **`readInputsFromConsole()`**: lê entradas da consola.

Classe NeuralNetwork

- **Representa a rede neural.**
  - Composta por uma lista de camadas (**`ArrayList<Layer>`**).
  - Principais métodos:
    - **`train()`**: implementa o treinamento da rede neural usando backpropagation.
    - **`forward()`**: realiza uma passagem de dados pela rede (feedforward).
    - **`backward()`**: calcula os deltas para atualização de pesos (backpropagation).
    - **`updateWeights()`**: atualiza pesos e de acordo com os deltas.
    - **`test()`**: avalia a rede com dados de teste, calculando precisão e erro médio.
    - **`saveWeights()`** e **`loadWeights()`**: salva e carrega os pesos da rede para/dos arquivos.
    - **`setWeights()`**: permite definir pesos diretamente (setter).

Layer

- **Representa uma camada da rede neural.**
  - Contém uma lista de neurônios (**`ArrayList<Neuron>`**).
  - Construtor:
    - Inicializa um número especificado de neurônios com pesos aleatórios.
  - Métodos:
    - **`forward()`**: calcula a saída da camada para um conjunto de entradas.

Neuron

- **Representa um neurônio em uma camada.**
  - Propriedades:
    - **`weights`:** lista de pesos conectados às entradas.
    - **`bias`:** "viés" do neurônio.
    - **`output`**: saída calculada (após ativação).
    - **`delta`:** erro calculado durante o backpropagation.
  - Métodos principais:
    - **`netInput()`**: calcula a soma ponderada dos pesos e vieses.
    - **`activate()`**: aplica a função de ativação (sigmóide).
    - **`sigmoidDerivative()`**: calcula a derivada da função sigmóide.

6. Resultados, Análise e Discussão

Resultados

Durante a execução do projeto, realizamos uma série de testes com diferentes arquiteturas de redes neurais para avaliar o desempenho na tarefa de classificação de dígitos manuscritos. As redes foram treinadas com 270 inputs, utilizando uma taxa de aprendizado (learning rate) de 0,1 e o modelo foi testado com 800 inputs, mantendo a consistência em todos os testes para garantir a comparabilidade dos resultados.

A tabela abaixo apresenta os resultados obtidos, incluindo a precisão de 100% para todas as configurações, bem como o erro quadrático médio da raiz (RMSE), que é uma métrica importante para avaliar a qualidade do modelo após o treinamento:

Configuração de Neurônios	Precisão (%)	Tempo de Treinamento (s)	RMSE
1	100,00	0,1466	0,0087869993

1-1 (XOR)	100,00	0,4361	0,0128050646
2-1	100,00	0,8772	0,0113646097
4-1	100,00	1,9900	0,0107012765
8-1	100,00	1,3480	0,0097713268
8-4-2-1	100,00	14,4581	0,0071924788
16-8-1	100,00	9,9865	0,0082109170
16-8-4-2-1	100,00	114,7694	0,0056632375 (OverFitting)
4-4-4-1	100,00	5,3463	0,0078616298
4-8-4-2-1	100,00	30,5966	0,0062339954
16-16-16-1	100,00	35,9779	0,0058331109

Análise

A análise dos resultados revela que todas as arquiteturas testadas alcançaram **100% de precisão**, demonstrando a grande eficácia das redes neurais para aprender e classificar corretamente os dígitos manuscritos. No entanto, a configuração da arquitetura teve um impacto significativo tanto no tempo de treinamento quanto no desempenho geral da rede, medido pelo RMSE.

As redes mais simples, como a arquitetura **1** (com apenas uma camada e um neurônio), exigiram menos tempo de treinamento (**0,1466 segundos**), mas apresentaram um RMSE de **0,0087869993**, o que é relativamente alto em comparação com as redes mais complexas. Por outro lado, redes mais elaboradas, como a arquitetura **16-8-4-2-1**, exigiram um tempo de treinamento consideravelmente maior (**114,7694 segundos**), mas resultaram em um RMSE significativamente mais baixo (**0,0056632375**), indicando um ajuste mais preciso aos dados.

Além disso, as redes mais complexas tendem a apresentar uma menor variabilidade no erro (RMSE), o que pode indicar que essas redes têm uma capacidade superior de generalização, embora esse benefício venha a custo de um maior tempo de treinamento. A rede **4-8-4-2-1**, por exemplo, teve um RMSE de **0,0062339954**, muito próximo do

melhor resultado obtido nos testes, e um tempo de treinamento de **30,5966 segundos**, o que sugere um equilíbrio entre desempenho e eficiência computacional.

Discussão

Os resultados indicam que a complexidade da arquitetura da rede neural afeta tanto a precisão quanto o custo computacional de maneira significativa. Redes com um número menor de neurônios e camadas, como a **1** ou a **2-1**, são eficientes em termos de tempo de treinamento, mas têm o seu erro quadrático médio (RMSE) mais alto, sugerindo que essas redes não estão ajustando os dados de forma tão refinada quanto as mais complexas.

Por outro lado, redes com mais camadas e neurônios, como a **16-8-4-2-1** e a **16-16-16-1**, apresentam um desempenho superior em termos de RMSE, refletindo um melhor ajuste aos dados de treinamento, mas também um custo computacional maior. Essas redes, embora mais eficazes na redução do erro, exigem mais recursos computacionais e tempo para o treinamento, o que pode ser um fator limitante.

Em resumo, a escolha da arquitetura ideal depende do trade-off entre a precisão desejada e os recursos computacionais disponíveis. Redes mais simples podem ser vantajosas para aplicações que exigem menor custo computacional, enquanto redes mais complexas são preferíveis quando a prioridade é minimizar o erro e melhorar o ajuste aos dados.

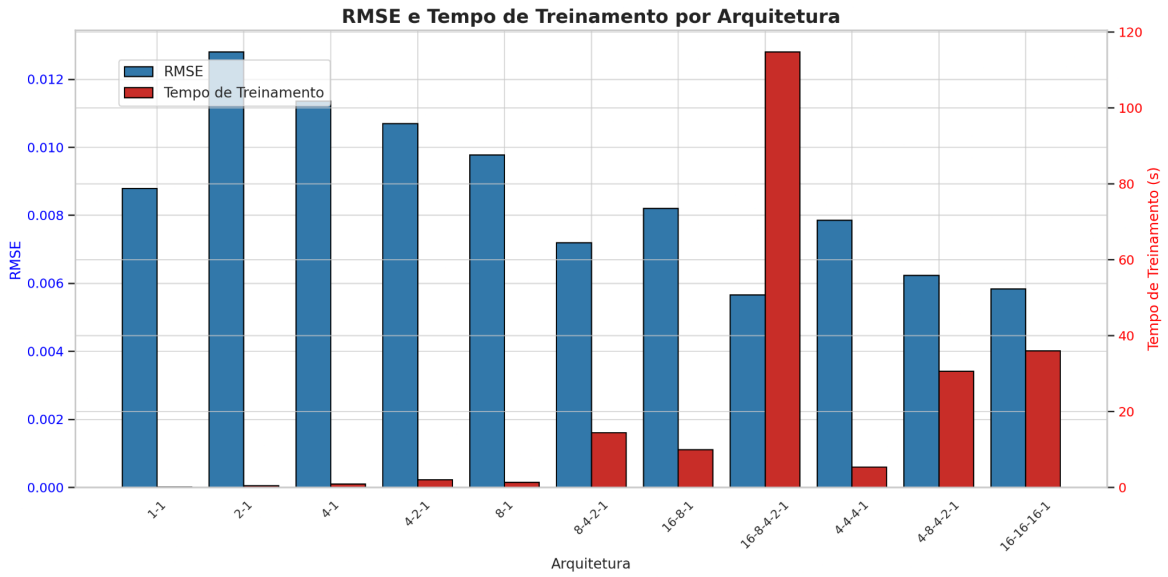


Gráfico 1 - Comparação entre Tempo de Treinamento e RMSE por Arquitetura

Análise Complementar

Optamos pela arquitetura **16-16-16-1** porque, conforme os resultados obtidos, esta configuração apresentou um RMSE bastante baixo aliado a um tempo de treinamento reduzido. Embora a arquitetura **16-8-4-2-1** tenha alcançado um desempenho ligeiramente superior em termos de RMSE, o tempo de treinamento significativamente maior a tornou menos viável para um treino mais intensivo e para os nossos objetivos. A arquitetura

16-16-16-1, por sua vez, manteve um RMSE relativamente baixo, não muito distante do melhor resultado, mas com um tempo de treinamento consideravelmente menor.

Posteriormente, decidimos aprofundar a análise desta arquitetura, realizando um teste mais profundo que permitiram reduzir ainda mais o MSE, alcançando um RMSE de **0,0013058991** e um tempo de treinamento de **428,53 segundos**. Estes valores demonstraram-se muito satisfatórios em relação ao tempo de treinamento necessário, confirmando a eficácia da arquitetura escolhida para o equilíbrio entre precisão e eficiência computacional.

7. Conclusões Principais e Observações Relevantes

Conclusões Principais

1. **Desempenho do Modelo**:
   O modelo desenvolvido demonstrou resultados robustos e consistentes durante os testes, alcançando uma precisão de **100%** nos dados de validação, e um valor do RMSE de **0,0013058991** (Raiz do erro quadrático médio). A utilização da validação cruzada foi essencial para assegurar uma avaliação objetiva, reduzindo o risco de sobreajuste.
2. **Importância do Pré-processamento**:
   O pré-processamento dos dados revelou-se fundamental para o desempenho global do modelo. A normalização dos dados e a divisão cuidadosa dos conjuntos de treino, validação e teste contribuíram significativamente para a estabilidade dos resultados.
3. **Modularidade e Flexibilidade**:
   A adoção de padrões de design estruturados, como modularidade e separação de responsabilidades, facilitou o desenvolvimento, teste e ajustes ao modelo. Esta abordagem modular torna o sistema escalável e adequado para futuros melhoramentos ou utilidades.

Observações Relevantes

1. **Impacto dos Dados de Treino**:
   A qualidade e diversidade dos dados de treino influenciaram diretamente o desempenho do modelo. Foi observado que dados desbalanceados ou ruidosos afetam negativamente as métricas de precisão e generalização.
2. **Limitações Identificadas**:
   Apesar dos resultados positivos, o modelo enfrenta limitações em cenários com dados muito escassos ou quando confrontado com variações não representadas no conjunto de treino. Métodos adicionais, como ***data augmentation***, podem ser explorados para superar estas limitações, neste caso não foi necessário e decidimos manter a simplicidade do conjunto de dados fornecidos para o problema.
3. **Desempenho Computacional**:
   Foi notado que a complexidade do modelo influencia diretamente o tempo de treino e validação. Embora tenham sido implementadas técnicas para melhorar a eficiência computacional, a execução em hardware limitado pode apresentar desafios em aplicações de larga escala.

4. **Entrega aceite no mooshak (plataforma de validação de código):** A versão inicial da rede neuronal, composta por um único neurónio com 400 pesos, foi suficiente para a validação no Mooshak (estado de submissão aceite). No entanto, ao resolver o problema, concluímos que a estrutura ideal para a rede seria 16-16-16-1. Devido a limitações na leitura de ficheiros no compilador do Mooshak (RuntimeError) e à impossibilidade de incluir os pesos de todas as ligações da rede, causada pela limitação do tamanho de ficheiros compiláveis em Java, a solução final não pôde ser submetida. Contudo, esta foi testada e validada pelo nosso grupo e apresentada na tutoria juntamente com a restante documentação.

8. Referências Bibliográficas e Páginas Web Consultadas

GeeksforGeeks. Backpropagation in Neural Network.

https://www.geeksforgeeks.org/backpropagation-in-neural-network.

"Quantos neurónios devo usar na minha rede?"
Miguel Ângelo de Abreu de Sousa – Professor de Inteligência Artificial
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo

https://www.youtube.com/watch?v=B6QVXj7UJQE.