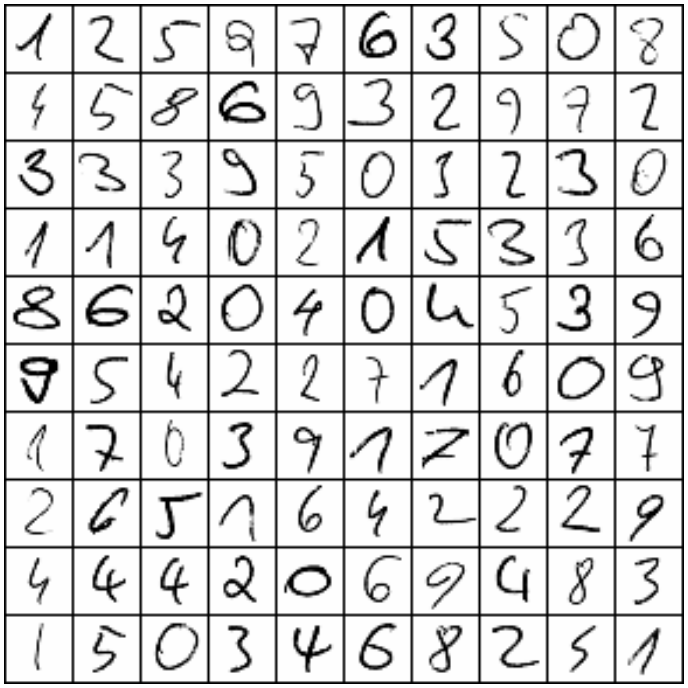
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

PROBLEMA 4
0 OU 1?

Detetar, a partir de uma imagem, se o número representado é 0 ou 1



ENGENHARIA INFORMÁTICA – INTELIGÊNCIA ARTIFICIAL

Trabalho realizado pelo grupo: IA2425P01G11

Data De Entrega: 9/12/2024

Índice

Introdução

A resolução deste problema foi realizada pelo Grupo 11 (P01), composto por Daniel Gonçalves (nº 79796), Diogo Damásio (nº 79826) e Gonçalo Rodrigues (nº 79833).

A aprendizagem automática está presente em qualquer ambiente digital que usamos. Estes algoritmos são usados para melhorar a nossa experiência de vida tanto pessoal, por exemplo sugerir músicas semelhantes às que gostamos, como profissional, por exemplo impressoras que foram ajustadas por algoritmos de aprendizagem para que não haja borrões na impressão. A aprendizagem automática é como a aprendizagem na vida de um humano, ou seja, o algoritmo aprende o que deve fazer com base em dados já classificados por humanos.

Muitas pessoas pensam que encontrar correlações entre dados é o que a aprendizagem automática faz, porém isso é apenas uma pequena parte da equação.

A maioria das tecnologias de aprendizagem são fiáveis, porém algumas, mesmo depois de receber treinamento, são constantemente imprevisíveis. É por este motivo que, para usarmos corretamente a aprendizagem automática temos de aprender como esta funciona. O entendimento da aprendizagem automática é necessário, pois esta está cada vez mais presente no mundo.

O problema 4 está relacionado com a aprendizagem automática, e implementa uma rede neuronal que distingue algarismos através de imagens. No entanto, este problema pretende uma simplificação dessa rede, uma vez que apenas será necessário distinguir entre os números 0 e 1, em vez de 0 a 9 como na rede original.

Este trabalho vai permitir compreender melhor como aplicar os princípios estudados nos trabalhos e aulas anteriores.

Descrição Do Problema e Algoritmos Utilizados

O problema 4 consiste em reconhecer o algarismo representado numa imagem de resolução 20 x 20 em *grayscale* que pertence à base de dados MNIST. Estas imagens estão representadas no ficheiro *dataset.csv*. O rótulo de cada imagem, ou seja, o número que representa, encontra-se no ficheiro *labels.csv*.

Seguem, em baixo, alguns exemplos da representação de algarismos utilizada neste problema:

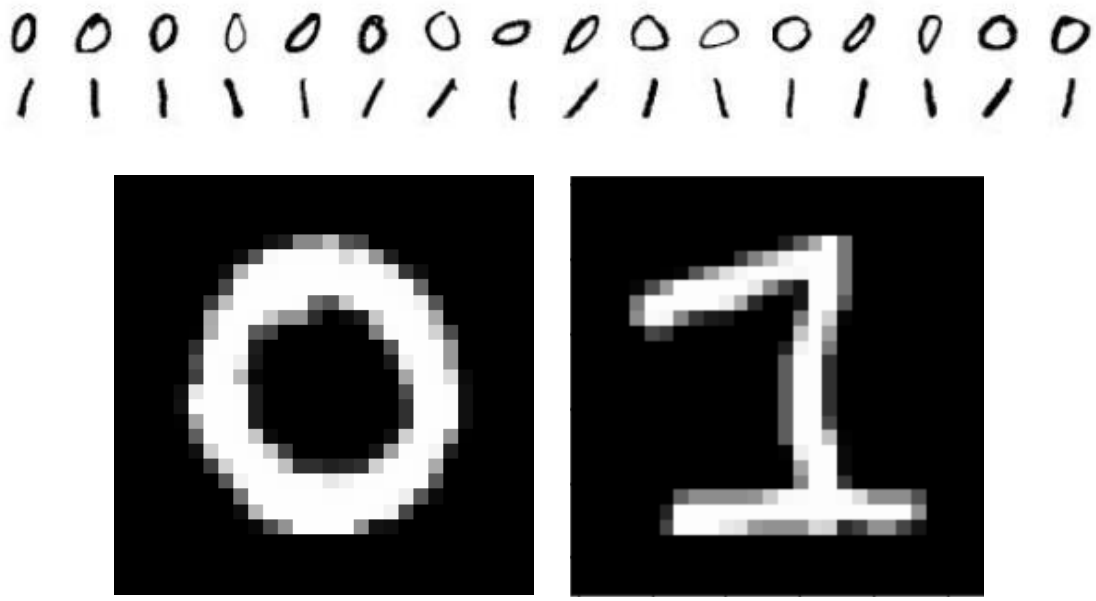


Figura 1 - Representação de algarismos, em grayscale

O conjunto de dados para este problema contém 800 imagens, cada uma correspondendo a uma linha no ficheiro. Cada linha contém 400 valores numéricos separados por vírgulas que representam o valor de cada pixel. Dito isto, existem 400 valores de entrada para a rede, cada um deles representando o valor de um pixel. Existe apenas um único valor de saída, 0 ou 1.

Do conjunto de dados disponibilizado, utilizámos 80% (640 imagens) para treinar a rede, e as restantes 160 para a testar, sendo que os elementos do conjunto de teste nunca foram utilizados para treinar a rede. Para além das imagens do conjunto de treino, considerámos também possíveis casos em que os algarismos pudessem ter rotações associadas. Assim, rodámos todas as imagens do conjunto de treino 90º no sentido horário, o que permitiu que fosse possível treinar a rede também para o caso em que os algarismos estivessem escritos “de lado”. Esta técnica denomina-se *data augmentation*, uma vez que cria “novos” dados a partir dos já existentes. No nosso caso, não consideramos que os dados gerados sejam novos, pois as imagens são as mesmas, apenas as rodámos.

Todos os valores numéricos de inputs tiveram de ser normalizados, ou seja, transformados em valores dentro do intervalo [0,1]. Ao analisar os valores no conjunto de dados fornecido, verificámos que a maioria já se encontrava neste intervalo. Os que ultrapassavam os limites do mesmo, tanto inferior como superiormente, eram, no entanto, próximos dos limites. Deste modo, decidimos que os valores que fossem

maiores que 1, seriam tomados como 1, e analogamente, se fossem menores que 0, seriam considerados 0.

A rede utilizada foi a rede composta por dois neurónios implementada no Problema 3, que tinha a finalidade de resolver a operação lógica XOR. A única alteração é o número de entradas, que passou a ser 400, e o número de pesos, que passou a ser 803 (400 do neurónio oculto, 401 do de saída e 2 *biases*). Optámos por utilizar esta rede porque, apesar de requerer um maior número de entradas e pesos, é uma rede simples que ainda assim é capaz de executar de forma eficiente e eficaz a tarefa de distinguir entre 0 e 1, que é o nosso objetivo principal.

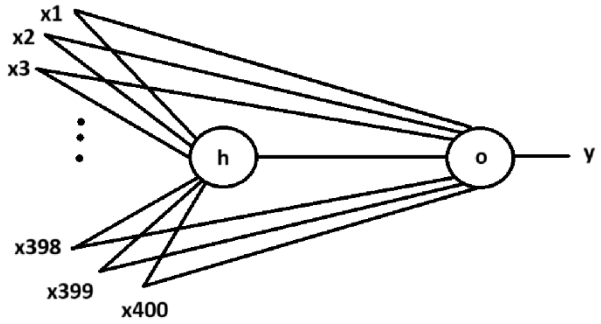


Figura 2 - Esquema da Rede Neuronal utilizada

Inicialmente o valor dos pesos e dos dois *biases* foi aleatoriamente definido no intervalo de 0 a 0,01, exclusivamente ([0;0,01[). Para treinar a rede, utilizámos o mesmo algoritmo de retro propagação de erro (na versão estocástica) descrito no Problema 3, com uma taxa de aprendizagem de 0,3. A função de custo utilizada será o erro quadrático, $\frac{1}{2}(saída\ esperada - saída\ obtida)^2$. Representá-la-emos por E.

Os próximos parágrafos irão resumir os cálculos efetuados para a atualização de cada peso, aquando da resolução do Problema 3.

Na expressão que permite atualizar um peso i, $\omega_i := \omega_i - \eta\Delta\omega_i$, o valor η é a taxa de aprendizagem, ω_i representa o peso e $\Delta\omega_i$ representa a sua variação. Este valor é dado por $\frac{\partial E}{\partial \omega_i}$. Se o representar a saída obtida da rede e z_o a ativação do neurónio de saída tem-se $\frac{\partial E}{\partial \omega_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z_o}\frac{\partial z_o}{\partial \omega_i}$.

Uma vez que $\frac{\partial E}{\partial o} = o - \hat{o}$, $\frac{\partial o}{\partial z_o} = o(1-o)$ e $\frac{\partial z_o}{\partial \omega_i} = x$, que é o valor da entrada para aquele peso, tem-se $\frac{\partial E}{\partial \omega_i} = (o - \hat{o})o(1-o)x$.

Sendo $(o - \hat{o})o(1-o) = \frac{\partial E}{\partial z_o} = \delta_o$, tem-se $\frac{\partial E}{\partial \omega_i} = \delta_o x$. Para os pesos da camada oculta, o raciocínio é semelhante $\frac{\partial E}{\partial \omega_i} = \frac{\partial E}{\partial o}\frac{\partial o}{\partial z_o}\frac{\partial z_o}{\partial h}\frac{\partial h}{\partial z_h}\frac{\partial z_h}{\partial \omega_i} = \delta_o \frac{\partial z_o}{\partial h}\frac{\partial h}{\partial z_h}\frac{\partial z_h}{\partial \omega_i}$.

Como $\frac{\partial z_o}{\partial h} = \omega_{ho}$ e representa o peso entre os dois neurónios, $\frac{\partial h}{\partial z_h} = h(1-h)$ e $\frac{\partial z_h}{\partial \omega_i} = x$, tem-se $\frac{\partial E}{\partial \omega_i} = \delta_o \omega_{ho} h(1-h)x$. Por sua vez, como $\delta_o \omega_{ho} h(1-h) = \frac{\partial E}{\partial z_h} = \delta_h$, tem-se $\frac{\partial E}{\partial \omega_i} = \delta_h x$.

À medida que o treino decorre, a rede tende a ficar demasiado adaptada aos valores que lhe são dados. Deste modo, quando lhe é apresentada uma imagem

diferente das usadas para treino, a rede pode não conseguir identificar o algarismo na imagem, e o erro quadrático médio (adiante denominado MSE) relativo ao conjunto de teste irá aumentar significativamente. Assim, torna-se necessário implementar um mecanismo que impeça que tal aconteça, interrompendo o treino se o MSE no conjunto de teste aumentar. Implementámos, então, um mecanismo de *early stopping*.

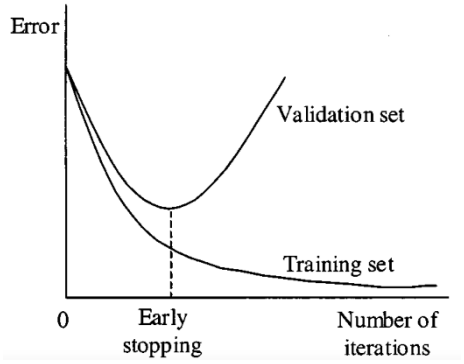


Gráfico 1 - Early Stopping

O mecanismo consistiu em monitorizar o MSE do conjunto de teste ao longo das épocas. Caso este subisse por 10 épocas consecutivas, o processo de treino parava. Desta forma, conseguimos garantir que a rede aprende o necessário das imagens do conjunto de treino, mas não excessivamente de forma a apenas conseguir dar uma resposta correta a essas imagens.

Para além deste critério de paragem, o processo de treino também parava caso o MSE do conjunto usado para treino fosse menor que 0,00001 ou 10^{-5}.

No fim do treino, os pesos finais foram impressos no terminal e serviram para a inicialização da mesma rede noutro programa. Este teve a finalidade de aplicar o treino anteriormente efetuado ao nosso problema de classificação: lê uma linha com os valores dos pixéis de uma imagem e determina se o algarismo representado é um 0 ou um 1.

Testes Unitários

Testámos os métodos que calculam previsões tanto de neurónios como de redes neuronais.

Para o efeito, reutilizámos os testes implementados no Problema 3, cujo conjunto de dados era a tabela de verdade da operação lógica XOR. Em cada teste, comparámos a saída da rede com a que seria esperada. O teste passava se todas essas igualdades fossem verdadeiras.

Abaixo encontra-se o teste unitário para a previsão do neurónio:

```java
/**
 * Testa o metodo predict
 */
@Test
void testPredict() {
    Neuron p = new Neuron(new double[] {1.0,1.0}, bias: -1.5);

    double obtido = p.predict(new double[] {0,0});

    assertTrue( condition: Math.abs(obtido-0.1824255238)<10e-9);

    obtido = p.predict(new double[] {0,1});

    assertTrue( condition: Math.abs(obtido-0.3775406688)<10e-9);

    obtido = p.predict(new double[] {1,0});

    assertTrue( condition: Math.abs(obtido-0.3775406688)<10e-9);

    obtido = p.predict(new double[] {1,1});

    assertTrue( condition: Math.abs(obtido-0.6224593312)<10e-9);

}
```

Figura 3 - Teste unitário para a previsão dos neurónios

De seguida, o teste unitário para a previsão da rede neuronal:

```java
/**
 * Testa o metodo predict
 */
@Test
void testPredict() {

    Neuron h = new Neuron(new double[] {1.0,1.0}, bias: -1.5);
    Neuron o = new Neuron(new double[] {1.0,1.0,-2.0}, bias: -0.5);

    NeuralNetwork nn = new NeuralNetwork(h,o);

    double obtido = nn.predict(new double[] {0,0});

    assertTrue( condition: Math.abs(obtido - 0.2963268202) < 10e-9);

    obtido = nn.predict(new double[] {0,1});

    assertTrue( condition: Math.abs(obtido - 0.4365732065) < 10e-9);

    obtido = nn.predict(new double[] {1,0});

    assertTrue( condition: Math.abs(obtido - 0.4365732065) < 10e-9);

    obtido = nn.predict(new double[] {1,1});

    assertTrue( condition: Math.abs(obtido - 0.5634267935) < 10e-9);
}
```

Figura 4 - Teste unitário para a previsão da rede neuronal

Resolução do Problema

1. Treino da Rede

É possível visualizar esta implementação em "Problema4\Implementações \TreinarARede".

No desenvolvimento deste programa, foram aplicadas diversas estratégias de design e padrões de projeto para garantir modularidade, reutilização e flexibilidade do código, promovendo também a facilidade de manutenção e expansão futura. Cada classe possui uma responsabilidade bem definida, como a classe Neuron, que encapsula o comportamento e as operações de um neurónio individual. Para reutilização, foram desenvolvidas funções genéricas, como as responsáveis pela leitura de dados de ficheiros, que podem ser reutilizadas em diferentes projetos.

A flexibilidade do código permite adaptações a diferentes problemas, e prevê, por exemplo, mudanças no tipo ou número de dados de entrada. Para além disso, a lógica de treino e a função de ativação podem ser modificadas sem comprometer a integridade do código existente.

Adicionalmente, foram utilizados padrões de projeto clássicos para estruturar o código. O padrão *Factory Method* foi utilizado para a criação de objetos específicos, como neurónios, permitindo que estes sejam facilmente modificáveis. O padrão *Observer* foi adotado para monitorizar o desempenho da rede durante o processo de treino, através do cálculo do erro quadrático médio em cada iteração de cada época.

Estas opções garantem um programa modular, escalável e preparado para evoluir, facilitando a adição de novas funcionalidades ou até alterações nas já existentes.

Classe Main

Esta é a classe responsável pela interação com o utilizador. Contém o método *main* para este fim, e *loadInputs* e *loadOutputs* para ler as imagens e a sua legenda, respetivamente.

Tem também os seguintes métodos:

- ***normalizeInputs*:** permite transformar os valores de input para valores no intervalo [0,1];
- ***augmentData*:** este método retorna um array que contém as imagens resultantes da técnica de *data augmentation*;
- ***augmentSample*:** este método aplica a técnica *data augmentation* a uma imagem;
- ***rotateImage90Degrees*:** método que roda a imagem fornecida 90º no sentido horário.

Resumidamente, a classe permite ler as imagens e as legendas, inicializar os neurónios com pesos aleatórios, e inicializar a rede com estes dois neurónios. Depois, inicia-se o processo de treino, e quando este se conclui, calcula-se a precisão da rede para o conjunto de teste (rácio de dados classificados corretamente). Por fim, imprime-se o valor dos pesos e *biases* para o terminal.

Classe Neuron

Esta classe manteve-se igual à implementada no Problema 3: tem o construtor de Neurons, *getters* e *setters* para os atributos, o método *predict* para prever a saída do neurónio, e a função *sigmoide* utilizada como função de ativação.

Classe NeuralNetwork

A classe também manteve a sua estrutura principal. Como já não era necessário, removemos o método *predictAll* implementado anteriormente, que previa a saída da rede para todos os inputs.

Adicionámos o método *precision*, que dado um conjunto de dados e respetivas saídas, calcula a precisão da rede. Para além deste método, também adicionámos o método *calculaErro*, que permite calcular o erro quadrático de um input dado a saída esperada. A implementação deste método permite evitar repetição de código no método *train*.

A principal alteração nesta classe foi mesmo no método para treinar a rede, e corresponde ao *early stopping*, como explicado anteriormente.

Diagrama UML

Este é o diagrama UML da implementação:

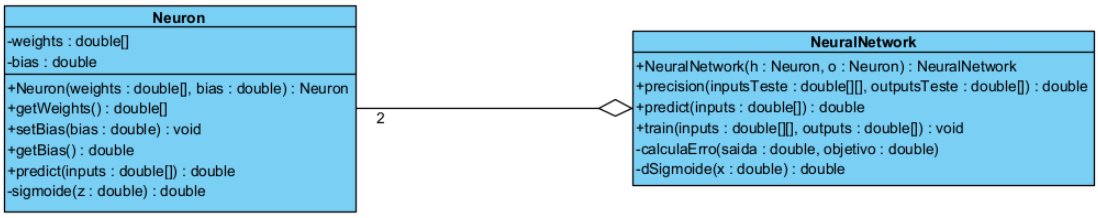


Diagrama 1 - Diagrama UML da implementação

JavaDoc

Consulte em "Problema4\JavaDoc\TreinarARede\index.html".

2. Rede treinada para classificação de algarismos

Esta implementação foi posterior à apresentada no ponto anterior e tem a finalidade de classificar o algarismo presente numa dada imagem. Utiliza, portanto, os pesos calculados na implementação anterior.

A implementação encontra-se em "Problema4\Implementações\RedeTreinada".

Classe Main

Em comparação com a anterior, nesta implementação apenas está presente o método *normalizeInputs*, idêntico, e o método *main*. No entanto, agora o método *main* processa uma imagem inserida pelo utilizador, e distingue-a como 0 ou 1, utilizando a capacidade de classificação da rede.

Classe Neuron

Esta classe mantém-se idêntica à da implementação anterior e às implementações do Problema 3.

Classe NeuralNetwork

Agora, esta classe tem apenas o construtor e o método *predict*, que calcula a saída da rede.

Diagrama UML

Este é o diagrama UML da implementação:

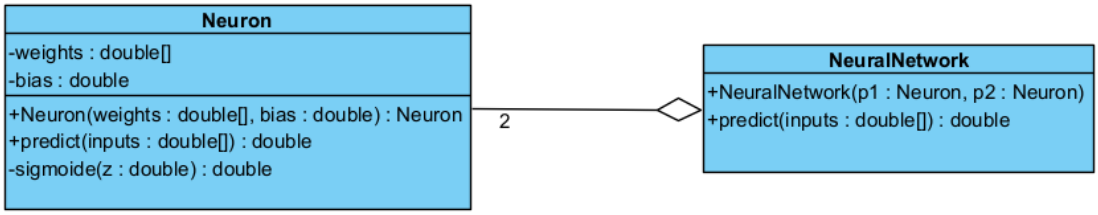


Diagrama 2 - Diagrama UML da implementação

JavaDoc

Consulte em "Problema4\JavaDoc\RedeTreinada\index.html".

Resultados e Análise

Corremos o programa algumas vezes e apresentamos aqui os melhores resultados. O processo de treino foi interrompido após 181 épocas, devido ao MSE de treino nesta época ser inferior ao limite que definimos. Quando terminou de treinar, o valor do erro quadrático médio de treino era de $9{,}9\times10^{-6}$, e o de teste $2{,}5\times10^{-5}$. A precisão para o conjunto de teste após o treino da rede foi de 100%.

Como o valor de MSE no final do treino é muito pequeno e a precisão é perfeita, é possível concluir que a rede aprendeu bem as características de ambos os algarismos, permitindo assim uma correta classificação das imagens.

De seguida apresentamos um gráfico que representa a evolução do erro quadrático médio (de treino e teste) ao longo das várias épocas do processo de treino.

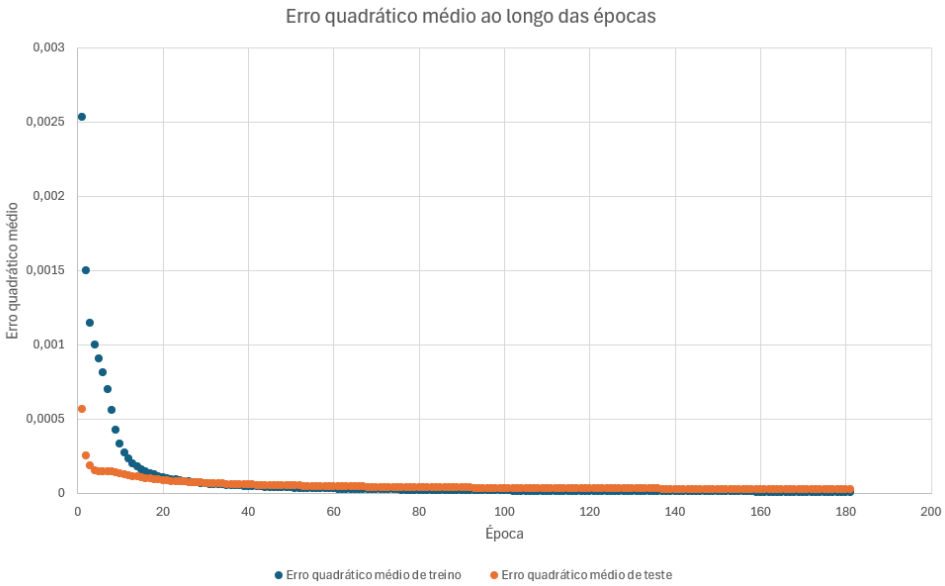


Gráfico 2 - Evolução do MSE ao longo do treino

Ambas as evoluções apresentam uma tendência logarítmica, decrescendo rapidamente no início e posteriormente estabilizando e decrescendo mais lentamente. Embora no início não aconteça, na generalidade do processo de treino (a partir da 28ª época) verifica-se que os valores de MSE de teste são ligeiramente superiores aos de treino, o que se comprova pela fina linha azul por baixo da amarela.

Tal ocorre porque a rede ajusta-se progressivamente às características das imagens utilizadas no treino, reduzindo continuamente o erro associado a este conjunto de dados. Contudo, quando a rede é submetida a imagens do conjunto de teste, cujas características não foram previamente retidas, a tarefa de classificação torna-se mais complicada, levando a um aumento do erro no conjunto de teste.

De realçar que, contrariamente ao previsto por nós, o MSE de teste nunca aumentou significativamente. Assim, o *early stopping* nunca foi ativado, e caso fosse retirado do código, os resultados obtidos não se alterariam.

Conclusões

Através da resolução deste problema, conseguimos obter esclarecimento acerca de como funcionam e como são implementados os algoritmos de classificação. Para além disso, aprofundámos o nosso conhecimento sobre técnicas já nossas conhecidas, como a retro propagação do erro, como também outras como *data augmentation* e *early stopping*.

Pudemos perceber que na generalidade do processo de treino, o erro quadrático médio de treino foi inferior ao de teste. Para além disso, pela observação do gráfico percebe-se que a estabilização foi rápida, o que evidencia que a rede identificou bem as características dos dois algarismos.

Consideramos ter tido sucesso na implementação deste problema, pois os erros obtidos foram pequenos e o número de épocas necessárias também foi relativamente pequeno. Para além disso, a precisão foi de 100%, tanto para o conjunto de teste por nós criado, como para o disponibilizado no Mooshak.

Referências Bibliográficas

Links:

- https://chatgpt.com/
- https://copilot.microsoft.com/
- https://www.w3schools.com/ai/ai_perceptrons.asp
- https://pages.cs.wisc.edu/~hasti/cs302/examples/UMLdiagram.html
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=wUBwMcuN5dD0k0zS
- https://cyborgcodes.medium.com/what-is-early-stopping-in-deep-learning-eeb1e710a3cf
- https://aws.amazon.com/what-is/data-augmentation/
- https://www.ibm.com/topics/data-augmentation

Outras referências:

- Slides da Cadeira – Inteligência Artificial;
- "Revolução do Algoritmo Mestre" – Pedro Domingos;

Ferramentas utilizadas:

- IntelliJ, IDE para programar em Java e gerar o JavaDoc;
- Visual Paradigm, para desenhar os UML's;
- Microsoft Word, para criar este relatório;
- Microsoft Excel, para obter os resultados do processo de treino criar o gráfico da evolução do MSE.