

# Compreensão de Linguagem Natural Americana

Pedro Escaleira

Departamento de Electrónica  
Telecomunicações e Informática  
*Universidade de Aveiro*  
Aveiro, Portugal  
escaleira@ua.pt

Rafael Simões

Departamento de Electrónica  
Telecomunicações e Informática  
*Universidade de Aveiro*  
Aveiro, Portugal  
rafaeljsimoes@ua.pt

**Abstract**—Serve este relatório como um resumo do estudo do comportamento de vários dos algoritmos mais utilizados em Aprendizagem Automática. Para isso, foi utilizado um *dataset* de fotografias de gestos de linguagem gestual americana e foi analisada e comparada a performance dos algoritmos Redes neurais, *Support Vector Machine* e Regressão Logística quando sujeitos a estes dados.

## I. INTRODUÇÃO

Este trabalho, proposto pela professora Petia Georgieva, do Departamento de Electrónica Telecomunicações e Informática da Universidade de Aveiro, teve o intuito de consolidar e instigar a utilização e pesquisa dos conceitos apresentados na disciplina de Tópicos de Aprendizagem Automática. Desta forma, estudamos o comportamento dos algoritmos **Redes Neuronais**, **Support Vector Machine** e **Regressão Logística**, de forma a perceber a influência que alguns dos parâmetros destes teriam influencia na performance dos mesmos e como solucionar possíveis problemas que advém da utilização de cada um.

## II. FERRAMENTAS USADAS

Este projecto foi executado usando-se *Python* como linguagem de referência. Contudo, para facilitar o nosso trabalho, usamos algumas bibliotecas *open source*:

- *Pandas*: usado para ler os dados dos *datasets* originalmente em *csv* para matrizes de *numpy*.
- *NumPy*: usado para fazer cálculos matriciais mais facilmente.
- *Matplotlib*: usado para criação facilitada de gráficos e figuras.
- *scikit-learn*: usado para estudar os algoritmos de aprendizagem automática descritos neste relatório.
- *Pickle*: usado para armazenar e carregar os modelos treinados em ficheiros binários.

## III. DADOS

As imagens usadas para fazer este estudo foram criadas, originalmente, por várias pessoas a reproduzirem múltiplas vezes os gestos referentes às letras pretendidas, com diferentes fundos, para uma maior diversidade de dados<sup>1</sup>. É possível

observar um exemplo destas fotos, já com algum pós processamento, na figura 1.

### A. Transformação do Dataset original

Pelo facto do *dataset* original conter um conjunto relativamente pequeno de dados (1704 imagens), este foi transformado num de maiores dimensões através de várias transformações feitas às imagens originais, sendo que o *dataset* obtido e com que trabalhamos possui um total de 34627 dados, distribuídos por 24 *labels*, numerados de 0 a 25 mapeados para as letras de A a Z alfabeticamente (foram excluídas as letras "J", correspondente ao *label* 9, e "Z", correspondente ao *label* 25, por necessitarem de movimentação gestual).

É de destacar também que os dados se encontram estruturados de forma similar aos do *MNIST*<sup>2</sup>: os objectos submetidos a classificação (neste caso, os gestos) estão centrados em imagens de 28x28 pixéis, em escala cinzenta, armazenados num ficheiro *csv*, onde a primeira coluna de cada linha corresponde ao *label* daquilo que está a ser classificado e as restantes colunas correspondem aos valores dos pixéis da imagem original, numa escala de 0 (preto) a 255 (branco), resultando num total de 785 colunas.



Fig. 1. Alguns exemplos de imagens, a cores, do *dataset*. Elas representam as letras do alfabeto americano em linguagem gestual que não necessitam de movimentação.

<sup>1</sup><https://github.com/mon95/Sign-Language-and-Static-gesture-recognition-using-sklearn>

<sup>2</sup><http://yann.lecun.com/exdb/mnist/>

Este tratamento realizado sob os dados originais não foi feito por nós, estando já disponíveis no *Kaggle*<sup>3</sup>.

### B. Divisão dos dados para o estudo

Apesar dos dados obtidos no *Kaggle* se encontrarem divididos em dois *datasets*, um para treino e outro para testes, seguindo o método *Hold-out*, decidimos, tal como sugeridos nas aulas da disciplina, dividir os dados em três partes, de forma a obter um terceiro *dataset*, para *cross validation*, de forma a fazer a selecção do melhor modelo e usar os dados de teste apenas para teste do melhor modelo que encontrámos para cada algoritmo. Desta forma, a divisão que decidimos efectuar sobre os dados foi a recomendada na disciplina, isto é, aproximadamente 60% (20776 dados) de dados de treino e 20% de *cross validation* (6925 dados) e de teste (6925 dados), distribuídos de forma aleatória por cada um a partir dos dados iniciais. Como é possível constatar pela figura 2, os dados ficaram distribuídos de forma praticamente uniforme por cada um dos *labels* existentes em qualquer um dos *datasets*.

### C. Forma como os dados foram explorados no estudo

Com os dados devidamente separados em três *datasets*, a forma como usamos cada um deles foi a seguinte:

- Os dados de treino foram usados para treinar os vários modelos que exploramos, mais concretamente.
- Os dados de *cross validation* foram usados para determinar o melhor modelo para cada variação de hiperparâmetros testada.
- Os dados de teste foram usados para perceber o quão boa a prestação modelo de cada algoritmo usado foi após fazer o *fine tuning* deste.

Para além disso, foi também usado o método de *Feature Scaling*, de forma a reduzir a escala das *features* (neste caso, os pixéis das imagens), dividindo cada uma delas por 255, já que é o valor máximo que um pixel pode possuir. Foi realizada esta normalização não com o intuito de trabalhar com todas as *features* na mesma escala, até porque já se encontravam, mas sim de forma a evitar *overflow* durante os cálculos realizados por cada algoritmo.

## IV. ESTUDO COM REDES NEURONAIAS

Sendo que o pressuposto das Redes Neuronais é que tenham um bom desempenho, em termos da qualidade da prestação de modelos treinados usando o mesmo, foi a primeira abordagem que decidimos dar ao problema. O maior problema que esperávamos encontrar ao usar este algoritmo era, como seria de esperar, a sua pobre prestação temporal a fazer o treino de cada modelo testado.

### A. Estrutura usada

Pelo facto de termos considerado que o problema não possuía uma complexidade extrema de ser resolvido, apresentando uma dificuldade similar ao do *MNIST*, onde a maior diferença se encontra no facto de neste caso haver uma maior

quantidade de classes para identificar, decidimos não usar um modelo de *deep learning*, mas sim uma rede de apenas 3 *layers*:

- Uma *input layer* com 784 nós.
- Uma *hidden layer* com 50 nós (foi um dos parâmetros que variamos, mas nos outros estudos, foi sempre este o tamanho usado).
- Uma *output layer* com 24 nós (sendo que a letra "Z" era uma das duas que não podia ser representada estaticamente e correspondendo a um *label* numa das extremidades, não foi contabilizada na *output layer*).

Para além disso, a inicialização dos *weights* foi feita de forma aleatória em qualquer modelo e os valores dos hiperparâmetros usados por *default* foram:

- $\lambda = 0$
- Número de iterações: 1000
- Activation Function: Logistic sigmoid function
- Batch Size: 10

Os resultados dos erros apresentados em cada um dos próximos tópicos foram obtidos através da formula do **Cross-Entropy Loss Function**, como é possível verificar na equação1.

### B. Estudo da variação do alfa

A primeira abordagem que decidimos ter ao testar o uso deste algoritmo foi a de variar o valor de alfa, isto é, o valor da *learning rate*, já que este parâmetro dita o tamanho de cada *step* tomado em cada iteração do *gradient descent*, afectando muito directamente a prestação que o modelo treinado apresenta, isto é, tem um papel fundamental no quão boa ou má é feita a generalização do modelo treinado em relação aos dados de *input*. Desta forma, sendo que no inicio não tínhamos noção de quais os melhores intervalos de alfa para este modelo, decidimos fazer um primeiro treino para valores mais desfasados de alfa. Os erros obtidos nestes modelos podem ser observados no gráfico da figura 3, onde se pode perceber que o menor erro foi obtido para um valor de alfa  $\alpha = 10^{-5}$ , pelo que percebemos que os melhores valores de alfa seriam encontrados em torno desse valor.

Sendo assim, decidimos treinar vários modelos para valores de alfa abaixo e acima de  $\alpha = 10^{-5}$ , mais concretamente, para valores de alfa entre  $5 * 10^{-6}$  e  $1.5 * 10^{-5}$  com intervalos de  $10^{-6}$  entre si. Os erros obtidos para cada um destes modelos podem ser consultados na figura 4.

O passo seguinte foi fazermos *retraining* do modelo que teve melhor prestação no passo anterior, ou seja, o que teve menor erro quando submetido aos dados de *cross validation*, neste caso para  $\alpha = 1.4^{-5}$ . Este passo foi essencial para diminuir ainda mais o erro e ter um modelo com uma ainda melhor prestação. Para isso, usamos o modelo já treinado, isto é, os *weights* obtidos no treino desse modelo e continuamos a treina-lo durante mais 10 000 iterações. Os valores da *cost function* ao longo das 11 000 iterações deste modelo podem ser consultados no gráfico da figura 5. Neste é impossível deixar de reparar que, no momento em que se iniciou o retreino

<sup>3</sup><https://www.kaggle.com/datamunge/sign-language-mnist>

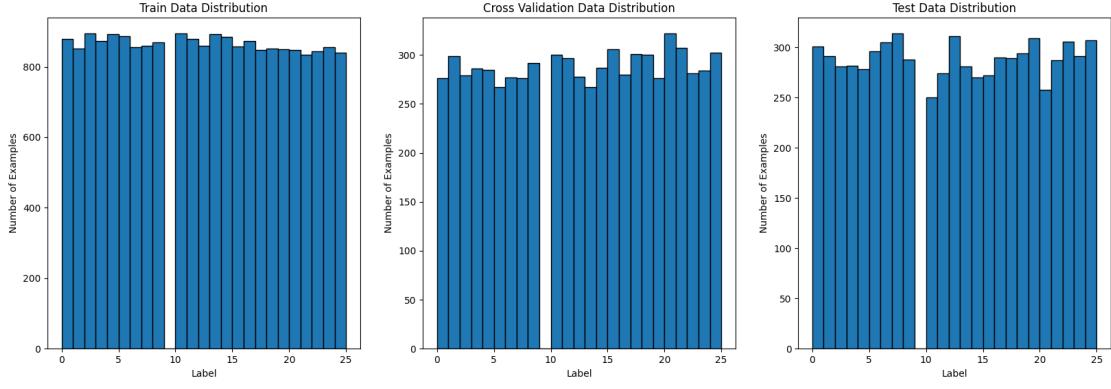


Fig. 2. Distribuição de dados por *label* por *dataset*.

$$E(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K [-y_k^i \log((h_\theta(x^i))_k) - (1 - y_k^i) \log(1 - (h_\theta(x^i))_k)] \quad (1)$$

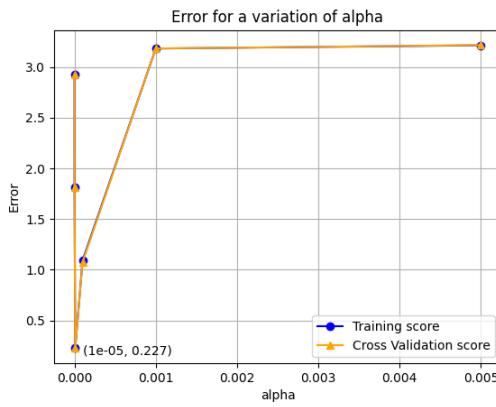


Fig. 3. Erro dos modelos obtidos para os valores de alfa  $\alpha \in \{10^{-3}, 5 * 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}\}$ .

do modelo e em algumas iterações após esse evento, houve um grande aumento do valor da *cost function*. Isto pode ser explicado pelo comportamento conhecido como *Catastrophic interference*, que é tido como o esquecimento total ou parcial daquilo que um modelo tinha aprendido num treino anterior a um novo treino com novos dados ou com novas classes [1]. Em termos de métricas, é possível verificar a performance do modelo perante os dados de teste, consultando os dados da tabela I. Duma forma geral, o modelo apresentou um valor aceitável em qualquer uma das métricas, sendo este valor igual para todas e igual a 97.7%.

Quanto aos valores da matriz de confusão deste modelo, com os dados de teste, podem ser consultados na tabela II. Nela, pode-se constatar que, de uma forma geral, o modelo praticamente adivinhou correctamente todas as classes.

### C. Estudo da variação do tamanho da Hidden Layer

Decidimos também, como parte do nosso estudo, verificar o comportamento deste algoritmo para a variação do tamanho da *hidden layer*, e verificar como o maior ou menor tamanho desta afectava a prestação do nosso modelo, bem como a complexidade computacional do treino do mesmo. Para isso, usamos o valor do melhor alfa obtido no ponto anterior e variámos o valor do número de nós da *hidden layer* de 5 em 5, entre 5 e 70. Assim sendo, obtivemos o gráfico dos erros de treino e de validação apresentado da figura 6.

Dos resultados obtidos, é-nos possível entender que o valor do erro para os dados de validação começa a estagnar para um número de nós acima de 40, inclusive. Uma possível explicação para este comportamento acontecer pode-se basear no facto de que, para um número de nós acima de 40, o custo do treino já praticamente convergiu. Como se pode perceber

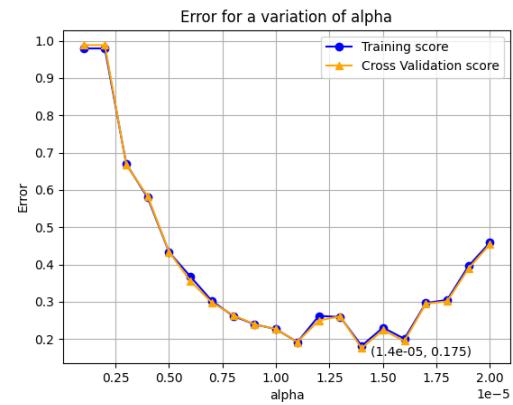


Fig. 4. Erro dos modelos obtidos para os valores de alfa  $\alpha \in \{\alpha \in R | 5 * 10^{-6} \leq \alpha \leq 2 * 10^{-5} | \alpha * 10^6 \in N\}$ .

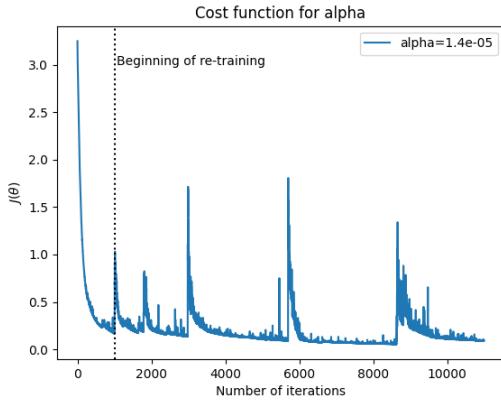


Fig. 5. *Cost function* para o alfa  $\alpha = 1.4 \times 10^{-5}$ , medida em cada iteração do treino do modelo.

TABLE I  
MÉTRICAS DE PERFORMANCE PARA  $\alpha = 1.4 \times 10^{-4}$

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	0.997	0.998
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	0.976	0.988
4	0.996	0.996	0.965	0.981
5	0.926	0.926	0.996	0.96
6	1.0	1.0	0.997	0.998
7	0.978	0.978	1.0	0.989
8	0.965	0.965	0.962	0.964
10	0.984	0.984	0.95	0.967
11	1.0	1.0	0.996	0.998
12	0.99	0.99	0.922	0.955
13	0.911	0.911	0.988	0.948
14	0.993	0.993	0.978	0.985
15	0.993	0.993	1.0	0.996
16	0.948	0.948	0.979	0.963
17	0.958	0.958	0.989	0.974
18	0.949	0.949	0.986	0.967
19	0.984	0.984	0.953	0.968
20	0.961	0.961	0.976	0.969
21	0.972	0.972	0.965	0.969
22	0.964	0.964	0.964	0.964
23	0.973	0.973	0.943	0.958
24	0.997	0.997	0.971	0.984
Macro Average	0.977	0.977	0.977	0.977

no gráfico da figura 7, para valores muito pequenos do número de nós, para 1000 iterações, a "rapidez" com que o custo converge é muito menor do que para valores mais elevados, sendo que para estes últimos, nas últimas iterações, os custos são muito parecidos de modelo para modelo. Fizemos também um estudo da complexidade temporal associado ao tamanho da *hidden layer*, já que este está directamente relacionado com a complexidade computacional do treino, já que quantos mais nós houver nesta camada, maior será o número de cálculos matriciais feitos. Como demonstrado no gráfico da figura 8, conclui-se que o tempo de execução varia, duma forma geral, linearmente de acordo com o número de nós. Por conseguinte, num caso de estudo onde se privilegia o tempo com que o treino é executado em detrimento de alguma qualidade nas

previsões feitas pelo modelo treinado, possivelmente seria aconselhável utilizar um número de nós mais reduzido, como por exemplo 40, obtendo-se um erro aceitável, mas claro, não tão baixo como seria se se usassem mais nós.

#### D. Estudo da variação do número de iterações

Ao obtermos o melhor valor de alfa, constatamos ao analisar o gráfico do custo de treino do melhor modelo ao longo das várias iterações que a *cost function* começava a convergir muito antes de completar 1000 iterações. Sendo assim, decidimos fazer um estudo da forma como o número de iterações iria afectar a performance do modelo. Na figura 9 é possível analisar o gráfico da evolução do erro de acordo com o número de iterações, para alfa  $\alpha = 1.4 \times 10^{-5}$ . Neste, pode-se constatar que para mais de 500 iterações, o valor do erro começa a ser parecido entre as várias iterações. Sendo que o número de iterações está intimamente relacionado com a complexidade computacional do algoritmo, como se pode verificar no gráfico da figura 10, chegamos à conclusão de que é possível obter modelos com uma performance aceitável em menos tempo, se executados durante menos iterações.

De acordo com vários estudos feitos, o método de *early stopping* é já usado em muitos estudos e projectos que usam redes neurais, dado o facto da complexidade computacional ser muito mais baixa e de automatizar o processo de selecionar o melhor modelo de acordo com o número de iterações necessárias para a curva da *cost function* começar a convergir [2]. Desta forma, decidimos experimentar treinar um modelo nas mesmas condições, mas usando este método, de maneira a verificar se conseguia-mos obter resultados aceitáveis. Neste caso, usamos um *stopping criteria* inverso do terceiro descrito no estudo [2], isto é, o treino acaba se, passado um determinado número de iterações consecutivas,  $nc$ , o valor da função de custo não melhorar mais do que um determinado valor predefinido,  $t$  (tolerância). Para  $nc = 30$ ,  $t = 10^{-4}$  e  $\alpha = 1.4 \times 10^{-5}$ , o modelo treinou durante 778 iterações e a performance obtida perante os dados de teste pode ser consultada na tabela III. Como seria de esperar, a performance

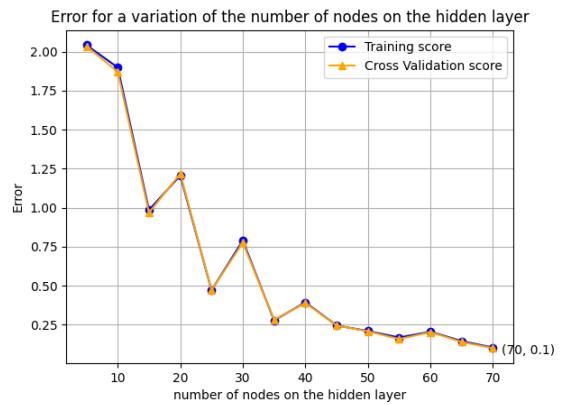


Fig. 6. Erro dos modelos obtidos para  $n \in \{n \in N \mid 5 \leq n \leq 70 \mid n/5 \in N\}$ , onde  $n$  representa o número de nós da *hidden layer*.

TABLE II  
MATRIZ DE CONFUSÃO PARA  $\alpha = 1.4 * 10^{-4}$

Class	0	1	2	3	4	5	6	7	8	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Predicted Class	301	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	301	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	291	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	281	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	282	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	277	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	274	0	0	0	5	0	0	0	0	0	0	0	0	5	0	5	0	7	0
6	0	0	0	0	0	0	305	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	1	307	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0	278	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0
10	0	0	0	0	0	0	0	0	4	246	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	274	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	308	0	0	0	1	0	2	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	22	256	0	0	3	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	2	268	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	270	1	1	0	0	0	0	0	0	0
16	0	0	0	5	0	0	0	0	0	0	0	4	0	6	0	275	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	277	0	0	6	0	6	0	0
18	0	0	0	0	10	0	0	0	1	0	0	0	1	0	0	1	0	279	0	0	0	0	0	2
19	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	304	0	0	0	4	0	0
20	0	0	0	1	0	0	0	0	0	8	0	0	0	0	0	0	1	0	0	248	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	279	5	0	3
22	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	295	6	4
23	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	5	0	283	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	306

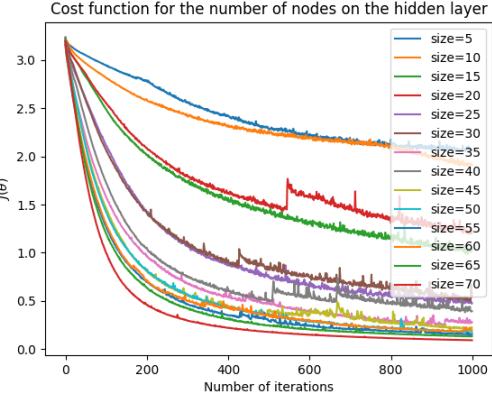


Fig. 7. Cost function para  $n \in \{n \in N \mid 5 \leq n \leq 70 \mid n/5 \in N\}$ , onde  $n$  representa o número de nós da *hidden layer*, medida em cada iteração do treino de cada modelo.

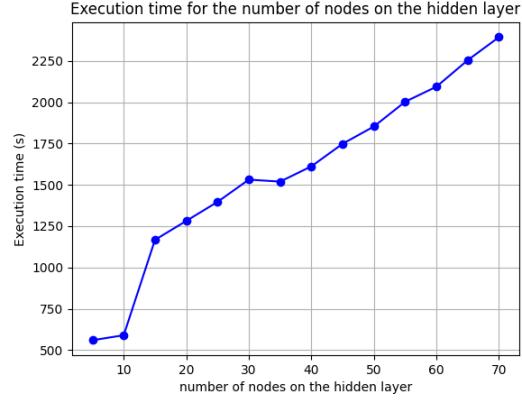


Fig. 8. Tempo de execução para  $n \in \{n \in N \mid 5 \leq n \leq 70 \mid n/5 \in N\}$ , onde  $n$  representa o número de nós da *hidden layer*.

não foi tão boa como para o modelo que usou 1000 iterações para o mesmo valor de alfa, mas pode ser aceitável em certos casos de uso, ainda para mais tendo em conta que o tempo de execução é muito mais reduzido do que fazer a totalidade de 1000 iterações.

#### E. Estudo da variação do tamanho da batch

Por ultimo, decidimos analisar como o tamanho da *batch* poderia influenciar a qualidade do modelo obtido. Para isso, usamos o melhor alfa já anteriormente obtido e treinamos os modelos durante 1000 iterações. De acordo com estudos já feitos nesta área, é possível aumentar o tamanho da *batch* durante o processo de treino ao invés da diminuição da

*learning rate* e, mesmo assim, obter resultados similares, mas num menor intervalo de tempo, já que um maior tamanho da *batch* implica menos actualizações dos parâmetros de treino [3]. Contudo, nós neste caso, e por uma questão de facilidade e limitações da ferramenta que usamos, o *scikit-learn*, fizemos um estudo para o treino de vários modelos com tamanhos de *batch* distintos entre si e constantes durante o treino, para um mesmo valor de alfa. O valor dos erros obtidos podem ser consultados no gráfico da figura 11. Numa primeira análise destes resultados, acha-mos estranho, já que esperávamos que o erro baixasse e não subisse, pelo menos para os dados de treino, ao aumentarmos o tamanho da *batch*. Contudo, ao analisarmos o gráfico da função de custo, verificamos que para

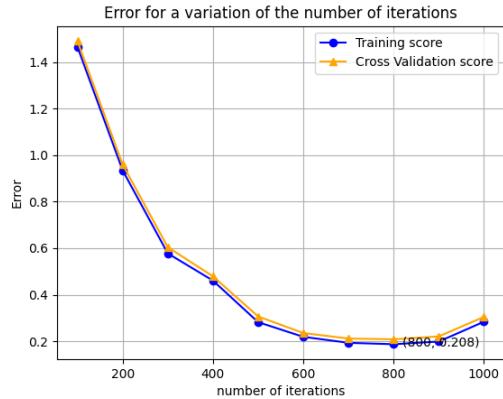


Fig. 9. Erro dos modelos obtidos para  $i \in \{i \in N \mid 100 \leq i \leq 1000 \mid i/100 \in N\}$ , onde  $i$  representa o número de iterações.

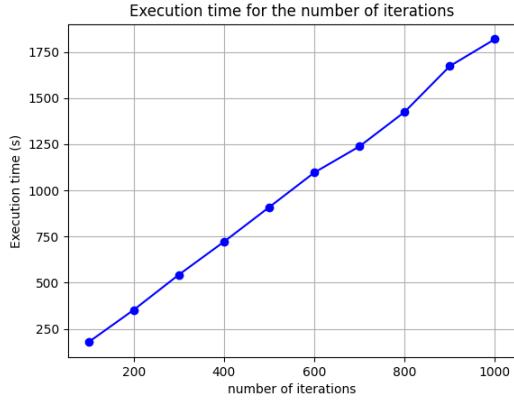


Fig. 10. Tempo de execução para  $i \in \{i \in N \mid 100 \leq i \leq 1000 \mid i/100 \in N\}$ , onde  $i$  representa o número de iterações.

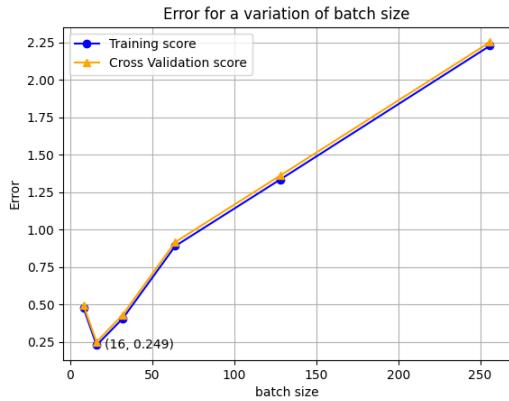


Fig. 11. Erro dos modelos obtidos para  $b \in \{b \in N \mid 8 \leq b \leq 256 \mid \sqrt{b} \in N\}$ , onde  $b$  representa o tamanho da batch usada.

um tamanho de *batch* mais elevado, a função convergia mais lentamente, como se pode verificar no gráfico da figura 12. Sendo assim, teríamos duas hipóteses: aumentar o número

TABLE III  
MÉTRICAS DE PERFORMANCE PARA UM MODELO CUJO TREINO USOU  
*early stopping* COM  $nc = 30$ ,  $t = 10^{-4}$  E  $\alpha = 1.4 * 10^{-5}$  E FOI  
EXECUTADO DURANTE 778 ITERAÇÕES

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	0.974	0.987
1	0.911	0.911	1.0	0.953
2	0.989	0.989	0.952	0.97
3	1.0	1.0	0.959	0.979
4	0.968	0.968	0.957	0.962
5	0.943	0.943	0.969	0.955
6	0.918	0.918	0.982	0.949
7	0.917	0.917	0.886	0.901
8	0.979	0.979	0.979	0.979
10	0.912	0.912	0.954	0.933
11	0.978	0.978	0.931	0.954
12	0.707	0.707	0.952	0.812
13	0.701	0.701	0.99	0.821
14	0.878	0.878	0.967	0.92
15	0.926	0.926	0.984	0.955
16	0.997	0.997	0.963	0.98
17	0.779	0.779	0.996	0.874
18	0.959	0.959	0.597	0.736
19	0.968	0.968	0.824	0.89
20	0.938	0.938	0.864	0.9
21	0.927	0.927	0.818	0.869
22	0.886	0.886	0.858	0.871
23	0.876	0.876	0.985	0.927
24	0.935	0.935	0.96	0.947
Macro Average	0.916	0.916	0.929	0.918

de iterações, o que não era pretendido neste caso, porque queríamos verificar se era possível obter um menor tempo de treino do que com *batchs* mais pequenas, ou aumentar o valor da *learning rate*, para um valor fixo do tamanho da *batch*, de forma a que a função de custo convergi-se mais rapidamente, tendo sido este o procedimento tomado e cujos resultados podem ser consultados no gráfico de erros da figura 13 e na tabela de métricas do modelo com menor erro (no *test dataset*) IV.

Apesar de não se ter obtido uma tão boa performance como quando utilizado um tamanho de *batch* mais pequeno (97.7% de *Accuracy* vs. 93.5% neste caso), é de notar que a performance é elevada e pode ser aceitável em muitos estudos, onde seja dada mais importância à rapidez com que o modelo é treinado. Como se pode verificar no gráfico da figura 14, a rapidez com que o treino é feito diminui substancialmente com o aumento do *batch size*.

#### F. Conclusões

Um ponto importante a destacar é que não fizemos qualquer estudo da variação do lambda<sup>4</sup>. Isto deve-se ao facto de não ter havido uma discrepancia muito notável entre os erros de treino e de *cross validation*, ou seja, não obtivemos qualquer tipo de *overfit*. Este problema poderá advir da *data augmentation* que foi feita aos dados originais, sendo que, ao os novos dados estarem distribuídos aleatoriamente pelos três *datasets*, tenha provocado que os modelos treinados tiveram contacto com todas as imagens originais, não existindo qualquer novidade

<sup>4</sup>parâmetro de regularização

TABLE IV

MÉTRICAS DE PERFORMANCE PARA  $batch\ size\ b = 256$  E ALFA  
 $\alpha = 9 * 10^{-5}$

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	0.98	0.99
1	0.979	0.979	0.966	0.973
2	0.996	0.996	1.0	0.998
3	1.0	1.0	0.986	0.993
4	0.917	0.917	0.992	0.953
5	0.932	0.932	0.962	0.947
6	0.954	0.954	0.942	0.948
7	0.949	0.949	0.99	0.969
8	0.962	0.962	0.982	0.972
10	0.948	0.948	0.868	0.906
11	0.945	0.945	0.827	0.882
12	0.932	0.932	0.924	0.928
13	0.936	0.936	0.916	0.926
14	0.981	0.981	0.996	0.989
15	0.908	0.908	0.888	0.898
16	0.938	0.938	0.925	0.932
17	0.893	0.893	0.819	0.854
18	0.874	0.874	0.918	0.895
19	0.919	0.919	0.937	0.928
20	0.919	0.919	0.919	0.919
21	0.857	0.857	0.908	0.882
22	0.83	0.83	0.951	0.887
23	0.893	0.893	0.935	0.914
24	0.98	0.98	0.929	0.954
Macro Average	0.935	0.935	0.936	0.935

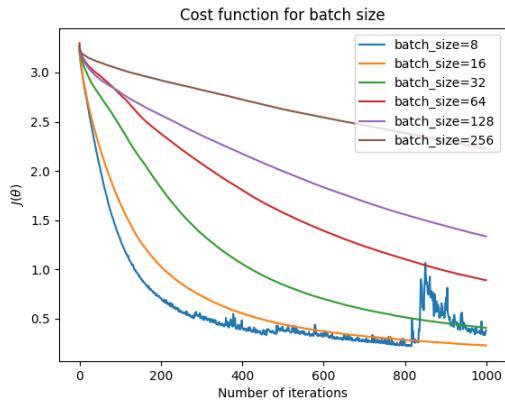


Fig. 12. Cost function para  $b \in \{b \in N \mid 8 \leq b \leq 256 \mid \sqrt{b} \in N\}$ , onde  $b$  representa o tamanho da batch usada.

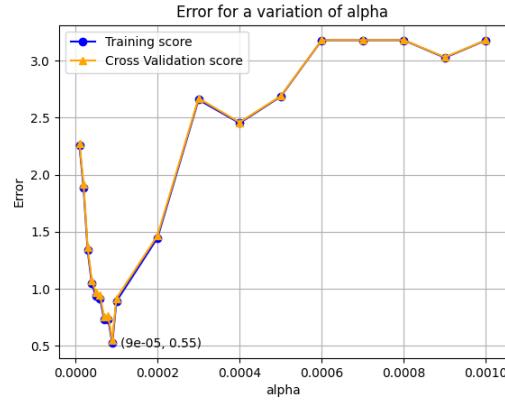


Fig. 13. Erro dos modelos obtidos para os valores de alfa  $\alpha \in \{\alpha \in R \mid 10^{-4} \leq \alpha \leq 10^{-3} \mid \alpha * 10^4 \in N\} \cup \{\alpha \in R \mid 9 * 10^{-5} \leq \alpha \leq 10^{-5} \mid \alpha * 10^5 \in N\}$  e para batch size  $b = 256$ .

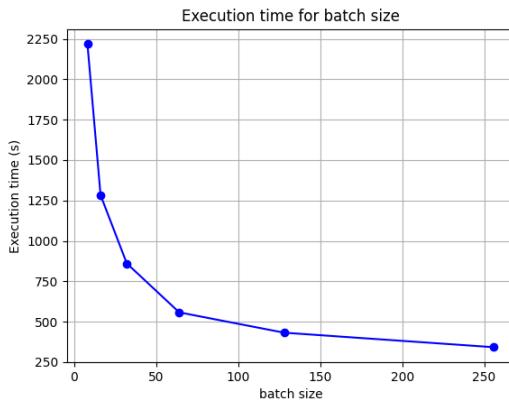


Fig. 14. Tempo de execução para  $b \in \{b \in N \mid 8 \leq b \leq 256 \mid \sqrt{b} \in N\}$ , onde  $b$  representa o tamanho da batch usada.

quando foi submetido aos dados de validação e de teste. Outro problema a apontar ao uso deste algoritmo é o facto da computação ser muito pesada, ainda para mais se feita com

recurso a um *CPU* ao invés dum *GPU*. A principal diferença entre usar um *CPU* e um *GPU* está no facto deste último usar uma muito menor quantidade de núcleos que o primeiro. Sendo que as computações mais exigente das Redes Neuronais são cálculos matriciais, que podem ser paralelizados, um *GPU* tem a capacidade de paralelizar uma muito maior quantidade de cálculos em simultâneo que um *CPU* pelo simples facto de possuir muitos mais núcleos para os fazer. Existem também outras razões para esses cálculos serem executados mais rapidamente em *GPsUs*, como a maior largura de banda e a maior dimensão e rapidez que os seus registos de memória possuem [4]. Sendo que usamos o *scikit-learn* para fazer o estudo e este apenas executa as computações em *CPU*, foi extremamente penoso proceder ao treino dos vários modelos apresentados. Chegamos por isso à conclusão de que, para este algoritmo, a resposta seja a utilização duma biblioteca como o *TensorFlow*, que permite a utilização do *GPU* do computador para treinar os modelos de Redes Neuronais. Em termos dos valores dos parâmetros a usar neste algoritmo para este problema, temos duas visões, que se baseiam no *trade off* que tem de ser feito entre a complexidade temporal e a qualidade dos resultados obtidos:

- Por um lado, se quisermos utilizar um modelo para um projecto onde seja privilegiada a qualidade dos resultados obtidos em detrimento do custo temporal que o treino do modelo necessita, então é recomendada a utilização dum valor de alfa reduzido, um batch size mais pequeno, um número de iterações elevado, sem *early stop* e uma *hidden layer* com um tamanho razoável (não muito elevado, para não ocorrer *overfit*).

- Por outro lado, se pretendemos usar o modelo num projecto onde seja privilegiado custo temporal e onde é aceitável obter resultados menos satisfatórios, chegamos à conclusão de que a utilização dum alfa com um valor mais elevado, com um tamanho de *batch* grande, com um número de iterações determinadas por uma regra de *early stop* e uma *hidden layer* de tamanho médio, levam à obtenção dum modelo que não possui a melhor performance possível, mas razoável e cujo tempo de treino é aceitável.

Claro que estas conclusões, apesar de para este problema em específico, podem ser generalizadas para outros problemas do mesmo género.

## V. SUPPORT VECTOR MACHINE (SVM)

SVM é um algoritmo simples, capaz de produzir taxas de sucesso elevadas com um poder computacional reduzido, apesar dos tempos de execução serem elevados devido à grande quantidade de *samples*. Após alguma pesquisa, confirmamos que SVM tem um desempenho bom para problemas de análise de imagens, muito pelo facto de este ser efectivo em espaços dimensionais grandes, que neste caso são o número de pixels de cada imagem (784). O objectivo deste algoritmo é encontrar um *hyperplane* (*decision boundary*) num espaço N-dimensional que distinga os dados fornecidos, onde será preferido o *hyperplane* cujo valor da margem seja maior. Devido ao que foi referido anteriormente, decidimos usar os *kernels* que nos pontos seguintes irão ser descritos.

### A. RBF Kernel

Antes de partir à análise desta *kernel* em específico, iremos fazer um pré balanço generalizado da mesma.

O parâmetro *gamma* define a região de influência que um determinado exemplo de treino atinge, tendo o significado de "longe" se o valor for baixo e "perto" se o valor for alto. O algoritmo é muito sensível a mudanças deste parâmetro, por isso é crucial que este seja o mais adequado possível, não podendo ser muito pequeno, com o risco de *underfit* (pois a região de influência de qualquer *support vector* irá incluir o *dataset* na totalidade) e não podendo ser muito elevado, com o risco de *overfit* (pois a região de influência de qualquer *support vector* irá incluir apenas o próprio *support vector*). Obviamente o que foi descrito em cima só é valido se hipoteticamente os valores de *gamma* pudessem ser  $(-\infty, +\infty)$ , respectivamente. O gráfico 15 ilustra a influência dos valores de *gamma*. Para complementar os dados definidos anteriormente, iremos mostrar um *heat map* 16 da *cross validation accuracy* em função de *C* e *gamma*. Como se pode visualizar, os valores que se encontram na diagonal tendem a providenciar um modelo com melhor *accuracy*.

#### 1) Valores default dos hiperparâmetros:

- $\gamma = 0.1$
- $C = 1$

Apenas de referir que a ordem de treino dos hiperparâmetros foi a seguinte:

- $\gamma$

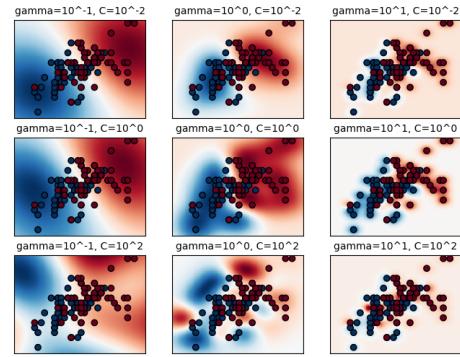


Fig. 15. Visualização da influência dos valores de *gamma*

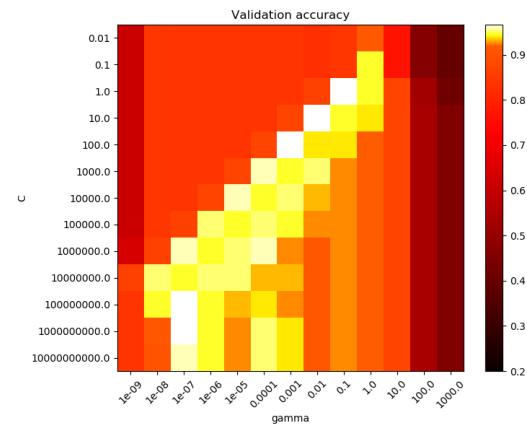


Fig. 16. HeatMap da Cross validation accuracy em função de *C* e *gamma*

- *C*
- 2) Equação da Gaussian radial basis function: 2

$$k(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2) \quad (2)$$

3) Estudo da variação do *gamma*: Definimos como primeira abordagem a atribuição de valores mais desfasados de *gamma*, sendo o leque de valores o seguinte: 3.

O erro do *train set* e *cross validation set* obtidos neste modelo para a variação do parâmetro *gamma* pode ser observados no gráfico 17. Como se pode observar ambos os erros têm valores muito próximos o que indica que o algoritmo obteve uma performance bastante boa.

Para complementar a análise feita a este algoritmo será apresentado o gráfico 18 de tempos das operações de *fit* e *predict*.

Com a análise destes dados podemos inferir que este algoritmo em particular tem uma performance excelente em todos os dados disponíveis, e que neste caso o critério de desempate foi o tempo de execução das respectivas operações de *fit* e *predict* que tendem a diminuir com o aumento da performance do algoritmo que neste caso corresponde ao aumento do valor de *gamma*. Isto justifica-se pelo facto de o algoritmo perder

muito tempo a tentar criar um *hyperplane* que se ajuste aos dados, não o conseguindo fazer eficientemente devido aos valores mal atribuídos dos hiperparâmetros. Podemos chegar à conclusão que o aumento de *gamma* permite a criação de um *hyperplane* mais complexo capaz de capturar a forma dos dados, o que não se verifica com valores mais pequenos pois a região de influência de cada *Support Vector* é maior causando maiores restrições no que toca à flexibilidade do modelo, causando o fenómeno de *underfit*. Por outro lado, se os valores de *gamma* forem muito elevados, o algoritmo produz um *hyperplane* demasiado complexo causando *overfit*, como é visível no gráfico 17, verificando-se o aumento drástico do *cross validation error*.

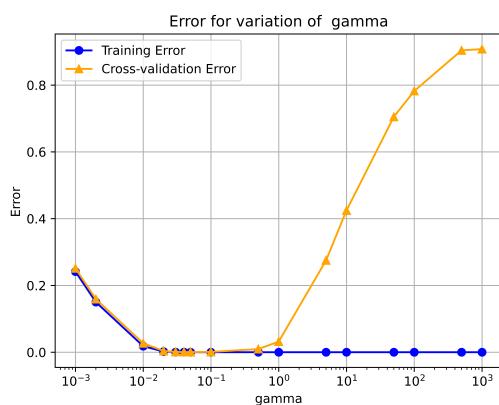


Fig. 17. Curva de validação para vários valores de *gamma*

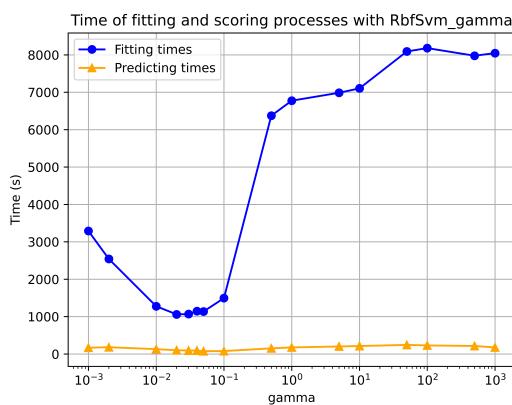


Fig. 18. Tempo das operações de *Fit* e *Predict* dos modelos para vários valores de *gamma*

#### 4) Estudo da variação do *C* (/*lambda*):

Apenas de referir que o melhor parâmetro de *gamma* foi usado no treino dos seguintes modelos, acompanhado o plano de selecção do melhor modelo. O parâmetro *C* comporta-se como um parâmetro de regularização para o *SVM*, na medida em que valores pequenos vão alargar a margem, ou seja uma função de decisão menos complexa será construída, verificando-se o oposto igualmente.

Acaba por ser um parâmetro capaz de definir o *trade-off* entre a correta classificação dos dados de treino e a maximização da margem da função de decisão. Mais uma vez optamos pela definição de um conjunto de valores desfasados, capazes de gerar modelos cujos estudos consigam ser conclusivos. Os valores para este parâmetro são  $C \in \{0.001, 0.002, 0.01, 0.02, 0.1, 0.2, 1, 5, 10, 50, 100, 500, 1000\}$ . Os erros de *train set* e *cross validation set* obtidos neste modelo para a variação do parâmetro *C* podem ser observados no gráfico 19 e os tempos de execução dos processos de *fit* e *predict* no gráfico 20.

Como se pode observar para valores demasiados pequenos de *C* o modelo acaba por não conseguir gerar um *hyperPlane* suficientemente complexo que corresponda à forma dos dados, causando o fenómeno de *underfit*. Mesmo para valores elevados de *C* o *Cross validation error* continua muito baixo acompanhado o erro do *Train Set*. Mais uma vez verifica-se que os tempos diminuem com o aumento da *accuracy* do modelo, pelos mesmos motivos explicados anteriormente.

5) *Conclusões:* Através do estudo feito à *SVM* com *kernel* gaussiana, podemos concluir que o algoritmo é mais sensível às variações do parâmetro *gamma*. Como se pode verificar no *heatmap* anterior, mesmo que os valores de *C* sejam ridiculamente altos a penalização na *accuracy* no modelo não é muito grande caso o *gamma* tenha os valores correctos, isto justifica-se pois este parâmetro actua como um parâmetro de regularização estrutural.

No final deste processo os melhores parâmetros foram os seguintes:

- *gamma* = 0.04
- *C* = 5

Para obtermos o melhor modelo possível, realizamos um último processo de treino com os melhores parâmetros, sendo agora apresentadas as tabelas de métricas de performance V e matriz de confusão VI.

#### B. Polynomial Kernel

Antes da análise, apenas de referir que todos os balanços feitos anteriormente relativamente aos parâmetros *C* e *gamma* aplicam-se a esta análise. Neste tipo de *kernel* mais um parâmetro é usado, *degree*. Este parâmetro é responsável pela flexibilidade da *decision boundary* em que valores mais elevados cria uma função mais complexa.

##### 1) Valores default dos hiperparâmetros:

- *C* = 1
- *degree* = 6
- *gamma* = 1 / nFeatures

##### 2) Equação da kernel polinomial: 4

$$r(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i * \mathbf{x}_j + 1)^d \quad (4)$$

##### 3) Estudo da variação de *C* (/*lambda*):

Os valores usados para este parâmetro foram  $C \in \{0.001, 0.002, 0.01, 0.02, 0.1, 0.2, 1, 5, 10, 50, 100, 500, 1000\}$ . Para ajudar na análise serão apresentados os gráficos de erro

$$\gamma \in \{0.001, 0.002, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500, 1000\} \quad (3)$$

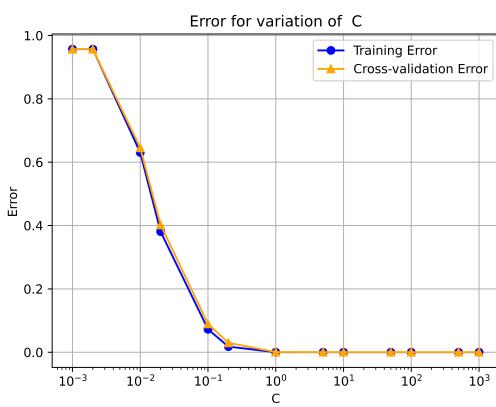


Fig. 19. Curva de validação para vários valores de  $C$

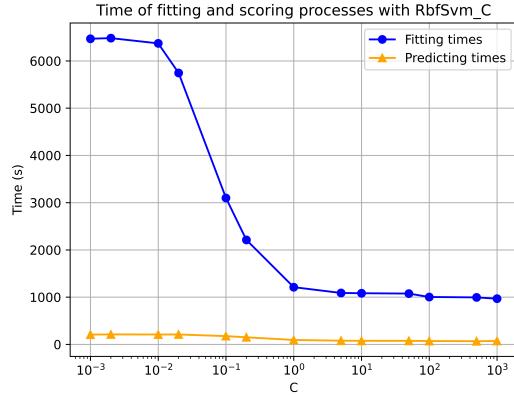


Fig. 20. Tempo das operações de *Fit* e *Predict* dos modelos para vários valores de  $C$

de *Train set*, *CV set*, gráfico 21, e os tempos de execução dos processos de *fit* e *predict*, gráfico 22.

Apesar de à primeira vista parecer que o fenómeno de *overfit* está a ocorrer, se analisarmos com mais detalhe podemos verificar que a diferença média entre os erros é muito baixa (0.002), o que indica uma boa performance destes modelos. Além disso, podemos verificar que os erros apresentados são bastantes constantes o que indica que o valor *default* de *degree* possa ter influenciado a consistência deste estudo. Comparativamente à *kernel* anterior, os tempos do processos de *fit* e *predict* são muito menores o que acaba por ser uma vantagem.

4) *Estudo da variação de degree*: Aqui definimos o leque de valores de *degree* como:  $degree \in \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}$ . Neste estudo foi usado o melhor parâmetro  $C$ . Como era de esperar, para valores muito grandes de *degree* o fenómeno de *overfit* tende a aparecer, pois uma função mais complexa acaba

TABLE V  
MÉTRICAS DE PERFORMANCE PARA  $\gamma = 0.04$  E  $C = 5$

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	1.0	1.0
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	0.996	0.996	1.0	0.998
5	1.0	1.0	1.0	1.0
6	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0
13	1.0	1.0	0.996	0.998
14	1.0	1.0	1.0	1.0
15	1.0	1.0	1.0	1.0
16	1.0	1.0	1.0	1.0
17	1.0	1.0	1.0	1.0
18	1.0	1.0	1.0	1.0
19	1.0	1.0	1.0	1.0
20	1.0	1.0	1.0	1.0
21	1.0	1.0	1.0	1.0
22	1.0	1.0	1.0	1.0
23	1.0	1.0	1.0	1.0
24	1.0	1.0	1.0	1.0
Macro Average	1.0	1.0	1.0	1.0

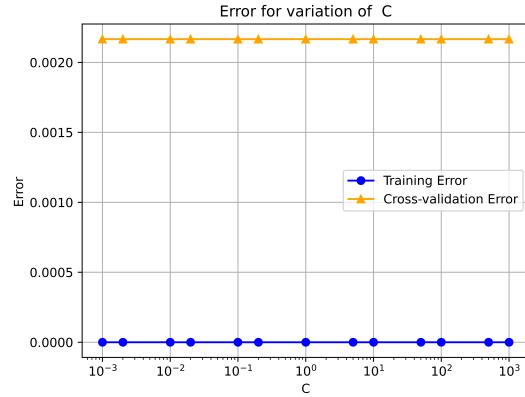


Fig. 21. Curva de validação para vários valores de  $C$

por ser criada moldando-se demasiado aos dados de treino não generalizando os outros casos. Isto pode-se comprovar com a visualização do gráfico 23, que apesar do erro estar a aumentar pouco, verificamos uma tendência de subida do *cross validation error* com o aumento exagerado do valor de *degree*.

5) *Notas*: Apenas de referir que apesar de neste tipo de *kernel* ser possível variar o parâmetro *gamma*, não o conseguimos fazer devido ao facto de a livraria que usamos para a realização deste estudo(*sklearn*) ter um tempo de execução muito elevado (maior que dois dias), justificável por esta trabalhar com o

TABLE VI  
MATRIZ DE CONFUSÃO PARA  $\gamma = 0.04$  E  $C = 5$

Predicted Class	Class	Actual Class																							
		0	1	2	3	4	5	6	7	8	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	0	291	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	281	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	282	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	277	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	296	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	305	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	314	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	288	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	250	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	274	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	311	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	281	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	270	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	272	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	290	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	289	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	294	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	309	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	258	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	287	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	306	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	291	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	307

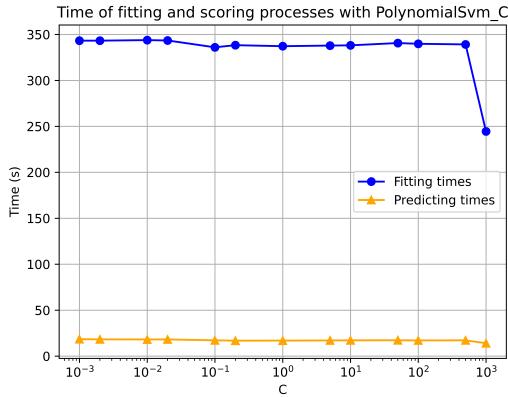


Fig. 22. Tempo das operações de *Fit* e *Predict* dos modelos para vários valores de  $C$

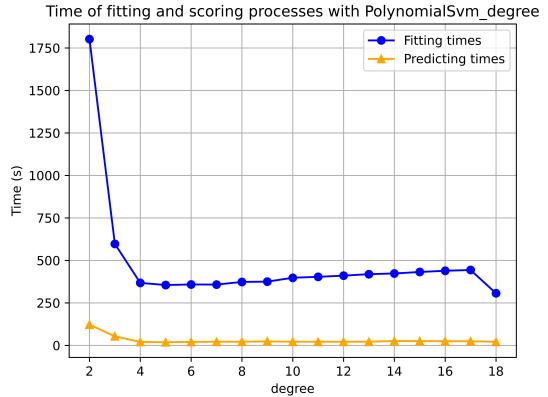


Fig. 24. Tempo das operações de *Fit* e *Predict* dos modelos para vários valores de  $degree$

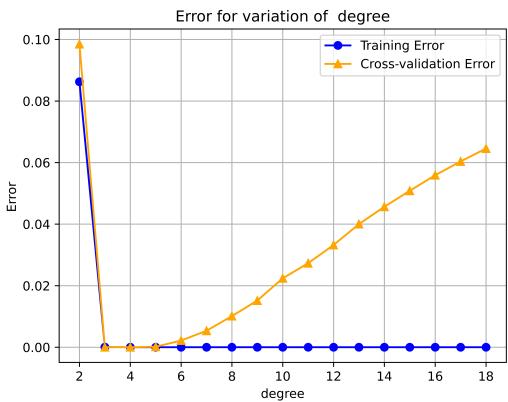


Fig. 23. Curva de validação para vários valores de  $degree$

*CPU* o que limita muito na execução de serviços necessários nos processos de *fit* e *predict*. Acreditamos mesmo assim, que as variações deste parâmetro tenham mais ou menos as mesmas implicações que na *kernel* apresentada anteriormente.

6) *Conclusões:* Com o estudo desta *kernel* foi possível verificar que este algoritmo mais uma vez é menos sensível às mudanças do parâmetro  $C$ , como se pode verificar no gráfico 21 ambos os erros mantêm-se inalterados muito por culpa do valor *default* do parâmetro  $degree$  (6). Apesar de este valor ter sido escolhido aleatoriamente verificou-se que com este valor o modelo tem um taxa de sucesso elevada, como é mostrado no gráfico 23 cujo o erro é praticamente 0, o que influenciou na consistência do gráfico 21. Mais uma vez se verifica o fenômeno de *overfit* caso os respectivos valores

do parâmetro *degree* sejam demasiado elevado, criando uma *decision boundary* demasiado complexa.

No final deste processo os melhores parâmetros foram os seguintes:

- $C = 0.01$
- $degree = 3$

Após o último processo de treino com os melhores parâmetros, obtivemos as seguintes tabelas: métricas de performance VII e matriz de confusão VIII.

TABLE VII  
MÉTRICAS DE PERFORMANCE PARA  $C = 0.01$  E  $degree = 3$

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	1.0	1.0
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0
6	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0
13	1.0	1.0	1.0	1.0
14	1.0	1.0	1.0	1.0
15	1.0	1.0	1.0	1.0
16	1.0	1.0	1.0	1.0
17	1.0	1.0	1.0	1.0
18	1.0	1.0	1.0	1.0
19	1.0	1.0	1.0	1.0
20	1.0	1.0	1.0	1.0
21	1.0	1.0	1.0	1.0
22	1.0	1.0	1.0	1.0
23	1.0	1.0	1.0	1.0
24	1.0	1.0	1.0	1.0
Macro Average	1.0	1.0	1.0	1.0

## VI. Multinomial logistic regression

Decidimos fazer um estudo com este modelo para esclarecer a eficácia do mesmo, que previamente achava-mos que ia ser baixa. Como se pode comprovar de seguida os resultados foram surpreendentes. *Multinomial logistic regression*, de entre os três estudados, é o algoritmo mais simples em termos matemáticos e de implementação. Este algoritmo consiste na generalização de *logistic regression* para problemas de multi-classes, assemelhando-se a uma função de pesos lineares. Posto isto é um modelo que é usado para calcular probabilidades de possíveis diferentes *outcomes*, tendo como valores de entrada um conjunto de variáveis independentes que podem ser de natureza real, binária ou categorial,etc.

### A. Valores defaults dos hiperparâmetros

- $C = 1$
- numero máximo de iterações = 1000

### B. Estudo da variação de $C$

Sabendo que este parâmetro se trata do inverso de lambda,ou seja é o parâmetro responsável pela regularização dos pesos para evitar fenómenos de *overfit*,foram definidos

os seguintes valores para a realização do estudo:  $C \in \{0.001, 0.002, 0.01, 0.02, 0.1, 0.2, 1, 5, 10, 50, 100, 500, 1000\}$ . No gráfico 25 pode se visualizar a curva de validação para vários valores de  $C$  e no gráfico 26 o tempo das operações de *fit* e *predict* dos modelos para vários valores de  $C$ . Como se pode visualizar, para valores de  $C$  baixos o modelo não consegue criar um função de decisão suficientemente complexa para lidar com a forma dos dados e por isso ocorre o fenómeno de *underfit*, que é justificável por ambos os erros serem elevados. Este erro começa a diminuir com o aumento de  $C$ , chegando a valores muito próximos de 0 com  $C$  geqslant 1. De todos os algoritmos estudados este acabou por ser o algoritmo com tempos de execução mais rápidos,o que era de esperar.

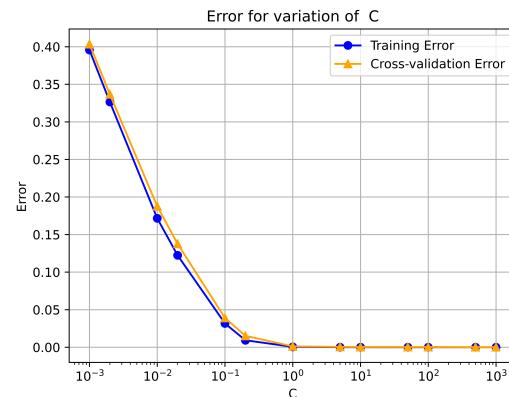


Fig. 25. Curva de validação para vários valores de  $C$

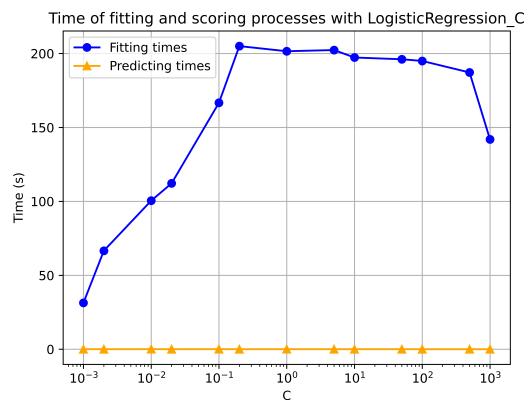


Fig. 26. Tempo das operações de *Fit* e *Predict* dos modelos para vários valores de  $C$

### C. Estudo da variação de numero máximo de iterações(*iter*)

Este parâmetro define o numero máximo de iterações que os solvers têm para conseguir convergir, tendo sido definido os seguintes valores:  $iter \in \{200, 500, 1000, 2000\}$ .

No gráfico 27 será apresentado a curva de validação para vários valores de iterações e no gráfico 28 os tempos de

TABLE VIII  
MATRIZ DE CONFUSÃO PARA  $C = 0.01$  E  $degree = 3$

Class	0	1	2	3	4	5	6	7	8	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Predicted Class	301	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	301	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	291	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	281	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	282	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	278	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	296	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	305	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	314	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	288	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	250	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	274	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	311	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	281	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	270	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	272	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	290	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	289	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	294	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	309	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	258	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	287	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	306	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	291	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	307

execução dos processos de *fit* e *predict*. Como era de esperar para um numero mais elevado de iterações a *accuracy* do algoritmo aumenta pois este tem mais iterações para chegar ao ponto de convergência, adversamente o tempo de execução do processo de *fit* aumenta.

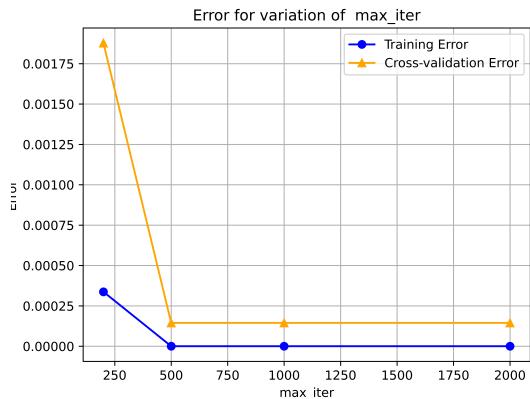


Fig. 27. Curva de validação para vários valores de *iter*

#### D. Conclusões

Com o estudo deste algoritmo foi possível concluir, que para o *dataset* em questão apesar de se tratar de imagens, este algoritmo simples conseguiu obter resultados impressionantes superando as nossas expectativas. Aqui o parâmetro  $C$  foi decisivo no resultado do modelo na medida em que ajusta os pesos da *decision boundary* possibilitando a criação de uma função mais complexa ou mais rígida. Neste caso para

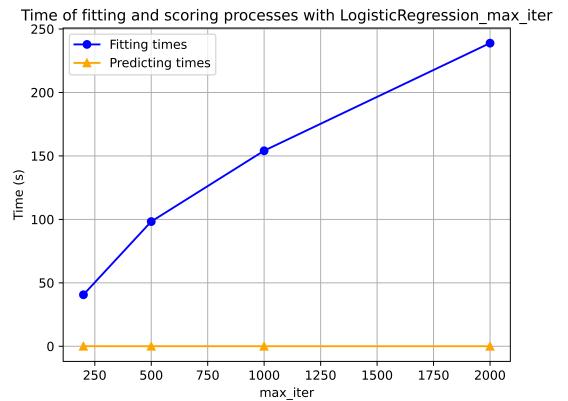


Fig. 28. Curva de validação para vários valores de *iter*

valores demasiados pequeno de  $C$  o modelo acaba por não conseguir generalizar os dados pois a função é demasiado simples, causando *underfit*, como é visível no gráfico 25. Para valores maiores e razoáveis de  $C$  a *accuracy* do modelo aumenta.

No final deste processo os melhores parâmetros foram os seguintes:

- $C = 500$
- $maxIter = 2000$

Após o último processo de treino com os melhores parâmetros, obtivemos as seguintes tabelas: métricas de performance IX e matriz de confusão X.

TABLE IX

MÉTRICAS DE PERFORMANCE PARA  $C = 500$  E  $maxIter = 2000$ 

Class	Accuracy	Recall	Precision	F1 Score
0	1.0	1.0	1.0	1.0
1	1.0	1.0	1.0	1.0
2	1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0
6	1.0	1.0	0.997	0.998
7	0.997	0.997	1.0	0.998
8	1.0	1.0	1.0	1.0
10	1.0	1.0	1.0	1.0
11	1.0	1.0	1.0	1.0
12	1.0	1.0	1.0	1.0
13	1.0	1.0	1.0	1.0
14	1.0	1.0	1.0	1.0
15	1.0	1.0	1.0	1.0
16	1.0	1.0	1.0	1.0
17	1.0	1.0	1.0	1.0
18	1.0	1.0	1.0	1.0
19	1.0	1.0	1.0	1.0
20	1.0	1.0	1.0	1.0
21	1.0	1.0	1.0	1.0
22	1.0	1.0	1.0	1.0
23	1.0	1.0	1.0	1.0
24	1.0	1.0	1.0	1.0
Macro Average	1.0	1.0	1.0	1.0

## VII. CONCLUSÕES

Finalizado este estudo, foi-nos possível consolidar o nosso conhecimento relativamente aos algoritmos mais populares usados em *Machine learning*, mais especificamente em processos de classificação multi-classe. O foco deste trabalho foi fazer um *Feature engineering* dos hiperparâmetros de entrada de cada algoritmo, baseando-se no processo sistemático de tentativa e erro de experimentar várias gamas de cada parâmetro, estudando-se os resultados e eficácia dos mesmos tentando perceber e justificar as respectivas performances com apoios em gráficos, outros estudos ou formulas matemáticas. Pelo facto de termos uma grande quantidade de dados, decidimos implementar e estudar algoritmos como redes neurais e *SVM* já que estes têm uma grande efectividade nestes casos. Foi possível concluir que especificamente para este *dataset*, usar algoritmos mais complexos como é o caso das redes neurais e *SVM* tornou-se *overkill*, uma vez que o algoritmo de *Logistic Regression* obteve óptimos resultados, muito melhores do que aqueles que esperávamos inicialmente. Isto pode ser justificado pelas mesmas razões do *MNIST* ser um *dataset* fácil de obter modelos que tenham uma boa performance: pelo facto de serem *datasets* simples, em que as imagens a serem analisadas se encontram em escala cinza, centradas e sem variações extremas, modelos tão simples como Regressão Logística têm a capacidade de gerar resultados com bastante qualidade. Para além disso, durante o estudo notamos que muitas vezes não houve *overfit* em modelos em que esperávamos ter acontecido. Como consequência, foi-nos impossível testar o papel da alteração do parâmetro de regularização nas Redes Neuronais, já que esta só é necessária quando há *overfit*. Isto poderá dever-se ao facto deste *dataset* ser o resultado de processos de *data*

*augmentation* sobre um *dataset* inicial de menores dimensões, pelo que quando os nossos modelos foram treinados, foram logo colocados sobre todos os possíveis dados, pelo que quando submetidos aos dados de teste e validação, estes não apresentaram muitas novidades.

## VIII. ANEXO

### A. Accuracy

$$\text{accuracy}(yi, yj) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(yj = yi) \quad (5)$$

### B. Recall

- $tp$  = True positives
- $fn$  = False negatives

$$\text{recall}(tp, fn) = \frac{tp}{(tp + fn)} \quad (6)$$

### C. Precision

- $tp$  = True positives
- $fp$  = False positives

$$\text{precision}(tp, fp) = \frac{tp}{(tp + fp)} \quad (7)$$

### D. F1 Score

$$f1(\text{precision}, \text{recall}) = 2 * \frac{(\text{precision} * \text{recall})}{(\text{precision} + \text{recall})} \quad (8)$$

## REFERENCES

- [1] Z. Chen, B. Liu, R. Brachman, P. Stone, and F. Rossi, *Lifelong Machine Learning: Second Edition*. Morgan & Claypool, 2018.
- [2] L. Prechelt, “Early stopping - but when?” 03 2000.
- [3] S. Smith, P. jan Kindermans, C. Ying, and Q. V. Le, “Don’t decay the learning rate, increase the batch size,” 2018. [Online]. Available: <https://openreview.net/pdf?id=B1Yy1BxCZ>
- [4] A. Kayid, Y. Khaled, and M. Elmahdy, “Performance of cpus/gpus for deep learning workloads,” 05 2018.

TABLE X  
MATRIZ DE CONFUSÃO PARA  $C = 500$  E  $\text{maxIter} = 2000$