Trabalho de Inteligência Computacional 23/24

Fase 3

Aprendizagem por transferência

Realizado por:

- Rodrigo Bernarda nº 2021136740 email: a2021136740@isec.pt
- Miguel Agostinho nº 2018016224 email: a2018016224@isec.pt

Coimbra, 13 de dezembro de 2023

Índice

Descrição do Problema

Algo que nos é ensinado desde cedo é a importância da reciclagem e o nosso dever sobre ela. Esta responsabilidade desempenha um papel essencial no desenvolvimento sustentável de uma cidade que se preocupa em preservar os seus recursos naturais e enfrentar os desafios ambientais que estão diante de nós.

Perante esta responsabilidade, temos como objetivo contruir uma rede neuronal capaz de identificar através de imagens RGB diversos tipos de lixo: plástico, metal, cartão, vidro e papel. Para esta fase utilizaremos aprendizagem por transferência de forma a melhorar a eficácia da nossa rede neuronal.

O dataset a utilizar é o *Garbage Classification* já utilizado nas fases anteriores, porém precisámos de alterar a estrutura do dataset visto que não se encontrava balanceado e portanto afetava a performance da rede neuronal. Dessa forma, o número de imagens de treino e teste por classe foram normalizadas para 800 e 160 imagens, respetivamente. Também decidimos remover a classe de 'lixo ordinário' para simplificar um pouco a rede neuronal.

Descrição das Metodologias utilizadas

Modelo de aprendizagem por transferência

Como mencionado anteriormente, para esta fase utilizaremos aprendizagem por transferência na arquitetura da nossa rede neuronal. Aprendizagem por transferência é uma técnica de *machine learning* que re-utiliza conhecimento prévio de um modelo já treinado numa tarefa para melhorar o desempenho de uma outra tarefa relacionada. Este tipo de modelos são também bastante úteis quando o conjunto de dados é limitado, não sendo necessário datasets com grande tamanho de imagens para atingir boas performances.

Rede Neuronal

```
base_model = Xception(weights='imagenet', include_top=False, input_shape=(71, 71, 3))

# Congelar camadas do modelo pré-treinado
for layer in base_model.layers:
    layer.trainable = False

callback = callbacks.EarlyStopping(monitor='loss', patience=3)

model = models.Sequential([

    base_model,
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(128, activation='relu'),
    layers.Dense(5, activation='softmax')
])

model.compile(optimizer= optimizers.Adam(learning_rate = 0.0001), loss = 'sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10, validation_split = 0.2, batch_size = 64)
```

Fig.1 – Rede Neuronal utilizada neste projeto

Existem diversos modelos de aprendizagem por transferência, como o Xception, o VGG16 e o InceptionV3. Para decidir qual modelo utilizaremos para o resto deste projeto, procederemos a uma comparação entre os 3 modelos.

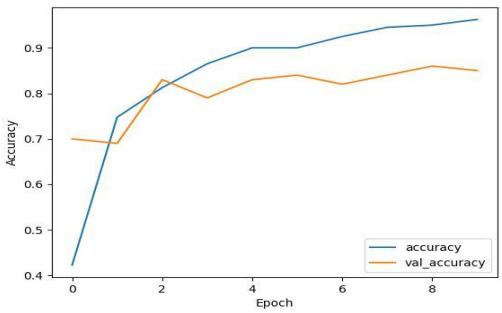

Fig.2 – Gráfico da accuracy de treino e validação do modelo Xception

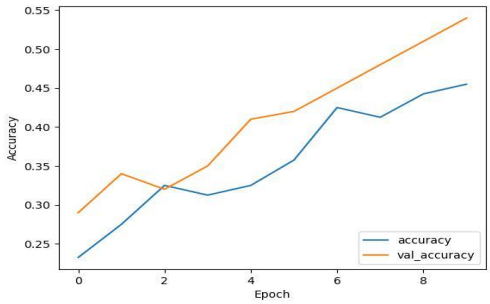

Fig.3 – Gráfico da accuracy de treino e validação do modelo VGG16

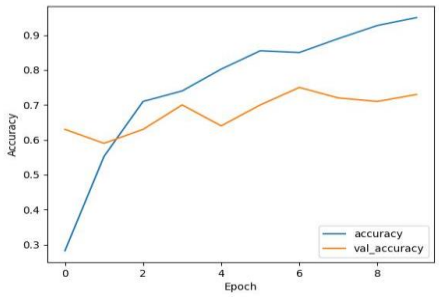

Fig.4 – Gráfico da accuracy de treino e validação do modelo InceptionV3

Para estes testes iniciais, foi utilizada, através do Keras, a arquitetura apresentada na Figura.1, constituida pelo modelo de aprendizagem por transferência, duas camadas de entrada densas de ativação ‘relu’, uma camada de dropout e uma camada densa de saída. No processo de compilação, é utilizado o otimizador ‘Adam’ com valor de learning rate de 0.001 e a função de perda 'sparse_categorical_crossentropy' e no processo de treino, são utilizadas 10 épocas, validation split de 0.2 e batch size de 64, o que acelera o treino da rede, sem comprometer a performance. Visto que modelos de aprendizagem por transferência são eficazes em datasets limitados, o tamanho do dataset foi reduzido para 100 imagens de treino e 20 de teste por classe, aumentando desta forma a velocidade de treino dos modelos. Como podemos observar, o modelo Xception foi o que obteve melhor performance diantes os outros modelos, chegando a uma accuracy de 0.86, acima da accuracy de 0.79 do InceptionV3 e muito acima da accuracy do modelo VGG16, que obteve resultados pouco satisfatórios, como se pode ver na Fig.3.

Camada de Dropout

Algo que estamos a utilizar pela primeira vez são as camadas de dropout. Dropout é uma técnica de regularização de redes profundas direcionada a prever overfitting, em que, em cada etapa, os neurónios das camadas de entradas serão temporariamente 'descartados', a partir de uma probabilidade habitualmente entre 0.1 a 0.5.

Nos testes inicias para encontrar o melhor modelo de aprendizagem por transferência foi utilizada uma camada de dropout com o valor de 0.5. Comparemos agora os resultados obtidos com o valor de dropout entre 0.2 e 0.5, já utilizando o modelo Xception.

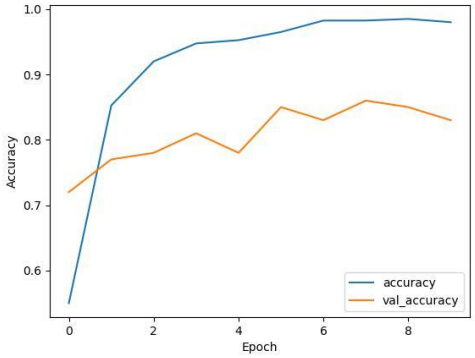

Fig.5 – Gráfico da accuracy de treino e validação utilizando dropout de 0.2

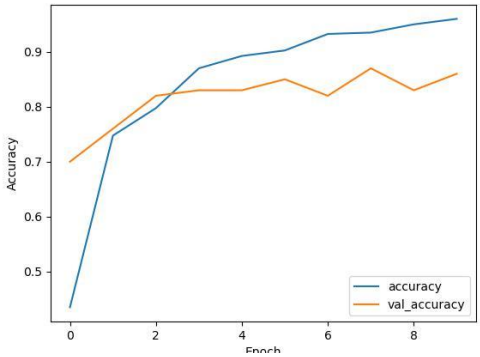

Fig.6 – Gráfico da accuracy de treino e validação utilizando dropout de 0.5

Como podemos observar na Figura.5, utilizando dropout de 0.2, as linhas de accuracy de treino e validação estão bastante afastadas, sendo um sinal claro que o modelo não está a ser capaz de generalizar bem para dados novos, portanto regista-se overfitting. Já utilizando dropout de 0.5, as linhas de accuracy de treino e validação encontram-se mais juntas, sendo então sinal que o modelo já está a ser capaz de generalizar para novos dados. Consoante estes resultados, para a próxima fase de otimização de hiper-parâmetros, a camada de dropout terá sempre um valor de 0.5.

Otimização de hiper-parâmetros

Tendo o modelo de aprendizagem por transferência e o valor do dropout já definidos, partiremos agora para a otimização dos hiper-parâmetros da rede neuronal. Decidimos escolher o número de neurónios nas duas camadas densas de entrada e a taxa de aprendizagem como os 2 hiper-parâmetros a otimizar. O objetivo desta otimização passa por maximizar o valor da accuracy do modelo. Para ajudar este processo, utilizaremos 3 diferentes técnicas de otimização:

- **Grid Search:** Grid Search é uma técnica de otimização de hiper-parâmetros, onde, dado um espaço de valores para cada hiper-parâmetros, o algoritmo realiza uma procura exaustiva através de todas as combinações desses valores para encontrar a configuração que otimiza o desempenho do modelo.
- **Randomized Search:** Randomized Search é uma outra técnica de otimização semelhante à Grid Search, porém em vez de percorrer todas as combinações possíveis de hiper-parâmetros, este algoritmo seleciona aleatoriamente um conjunto específico de combinações a avaliar. O objetivo passa explorar um subconjunto aleatório do espaço de hiper-parâmetros em vez de examinar todas as combinações possíveis, sendo particularmente útil em situações em que o espaço de procura é grande, economizando desta forma tempo computacional.
- **Particle Swarm Optimization (PSO):** PSO é um algoritmo de otimização swarm que se inspira no comportamento social de enxames. Neste algoritmo, uma população de partículas move-se pelo espaço de procura, ajustando as suas posições com base na experiência individual e coletiva. Comparando com os 2 algoritmos acima descritos, o PSO é um algoritmo que tende a enfocar a exploração global do espaço de soluções, movendo-se em direção a regiões promissoras, enquanto o Grid Search e Randomized Search são mais orientados para a exploração local, cobrindo sistematicamente todo o espaço de procura. Na fase anterior estudámos o Firework Algorithm, porém optámos por utilizar o PSO para esta tarefa de otimização dada a sua simplicidade e eficácia neste tipo de tarefas.

Apresentação da Arquitetura do Código

Para a elaboração deste projeto, utilizamos 3 ficheiros, mais um ficheiro *historico.ipynb* onde é possível carregar e avaliar modelos anteriormente treinados. Passemos agora à apresentação das funcionalidades dos 3 ficheiros utilizados.

- **Projeto.ipynb:** Este é o ficheiro principal utilizado, onde é carregado e normalizado o dataset, treinada e avaliada a rede neuronal. No diagrama abaixo é apresentado o funcionamento deste ficheiro:

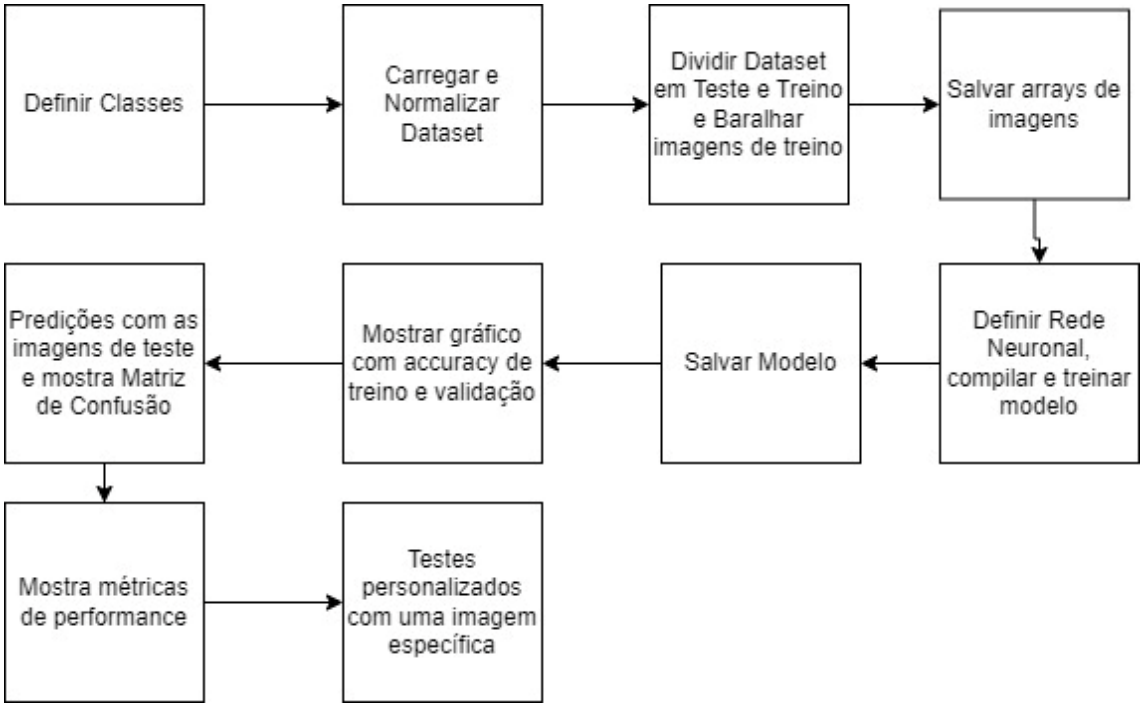

Fig.10 – Desenho da Arquitetura do projeto.ipynb

- **Otimizacao.ipynb:** Neste ficheiro é realizada a otimização da rede neuronal através dos algoritmos Grid Search e Randomized Search. No diagrama abaixo é apresentado o funcionamento deste ficheiro:

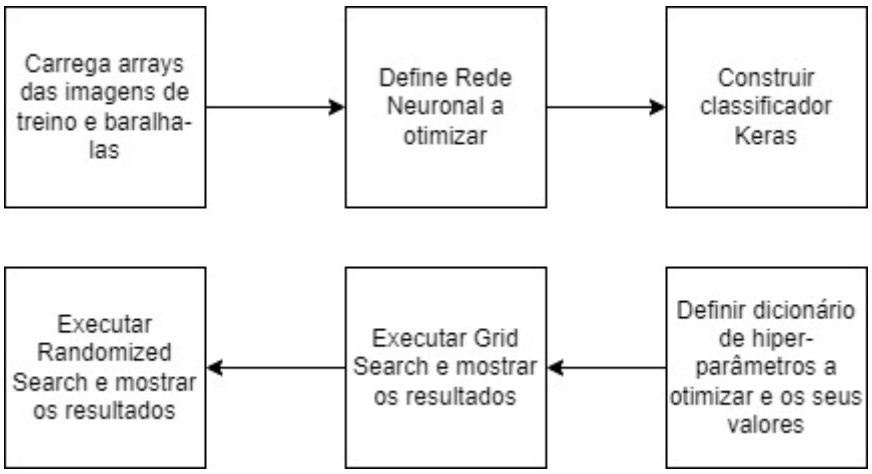

Fig.11 – Desenho da Arquitetura do otimizacao.ipynb

- **Swarm.ipynb:** Neste último ficheiro é realizada também a otimização da rede neuronal, porém utilizando o algoritmo de otimização PSO. No diagrama abaixo é apresentado o funcionamento deste ficheiro:

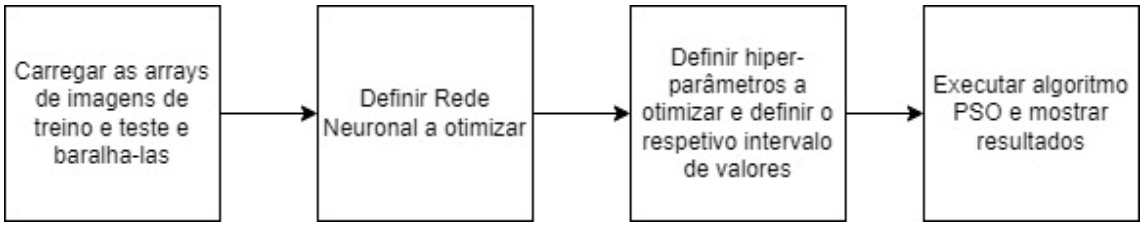

Fig.12 – Desenho da Arquitetura do swarm.ipynb

Descrição da Implementação dos algoritmos

Projeto.ipynb

Como referido no capítulo anterior, começamos primeiro por definir o nome das classes e a sua respetiva label. Depois utilizamos a função *load_data()* que carregará as pastas das imagens de treino e teste do dataset, redimensiona e normaliza-las. Ao chamar a função divide-se o dataset em arrays de imagens de treino e teste, e usa-se a função *shuffle* da biblioteca sklearn.utils para baralhar as arrays de imagens de treino, de forma a evitar que os dados não sejam treinados numa ordem específica, e são salvas, através da função *save* da biblioteca numpy, as arrays de forma a serem usadas nos outros ficheiros. Após este processo, é definida, através da biblioteca Keras, a rede neuronal da Fig.1, compilada e treinada como referido no Capítulo 1, e no final é salvo o modelo e histórico de treino. Utilizando os dados do histórico de treino e a biblioteca matplotlib.pyplot é criado um gráfico com os dados da accuracy de treino e validação e modelo. No final, é utilizado o método *predict* para fazer previsões nas imagens de teste utilizando o modelo treinado, e com isso usa-se as funções de métricas da sklearn para criar uma matriz de confusão (utilizando também a biblioteca seaborn para criar um

mapa de calor) e devolver métricas de desempenho, como accuracy, precision, f-measure e AUC.

Otimizacao.ipynb

Neste ficheiro, são primeiro carregadas através da função *load* da biblioteca numpy as arrays das imagens de treino e são novamente baralhadas usando a função *shuffle* da sklearn. Depois é definida a função *keras_model* que devolverá o modelo compilado para otimizar. Utilizando a função *KerasClassifier* da biblioteca scikeras, é possível então utilizar o modelo Keras definido na função *keras_model* com as ferramentas de Grid Search e Randomized Search da scikit-learn, servindo assim como ponte entre o Keras e Scikit-Learn. Como referido no capítulo 1, os hiper-parâmetros que decidimos otimizar são o número de neurónios das camadas densas de entrada e o learning rate do otimizador, como tal definimos 3 valores a testar para estes 2 hiper-parâmetros: o número de neurónios variará entre 64, 128 e 256 neurónios e o valor do learning rate variará entre 0.0001, 0.001 e 0.01. Depois de definido a gama de valores dos hiper-parâmetros é executado o Grid Search e o Randomized Search, sendo depois imprimido os valores médios da accuracy e o seu desvio padrão entre cada combinação de hiper-parâmetros. Em ambos algoritmos é utilizada uma estratégia de cross-validation de 5 folds, o que significa que o dataset será dividido em 5 partes e o modelo será treinado e avaliado 5 vezes, sendo usada uma parte diferente do dataset em cada iteração. Este método permite uma avaliação mais estável e representativa do modelo.

Swarm.ipynb

Semelhante ao ficheiro anterior, começamos por carregar as arrays das imagens de treino e teste e baralhamos-las. Depois criamos a função *fitness_function* onde será definido o modelo Keras a otimizar, assim como compilado, treinado e por fim avaliado com as imagens de teste, pois ao contrário da função *keras_model,* a função *fitness_function* retorna o valor de 1 menos a accuracy, assim o é pois o algoritmo PSO tem como objetivo minimizar um certo valor, então para maximizar o valor da accuracy temos que devolver ao algoritmo o valor subtraido da accuracy. Após definir esta função, definimos os intervalos de procura de soluções para o algoritmo PSO, que serão neste caso a gama de valores possíveis do número de neurónios ([64,128]) e o valor do learning rate ([0.0001,0.001]). Por fim é executado o algoritmo PSO, utilizando 3 partículas, a função a otimizar *fitness_function*, os intervalos definidos e 10 épocas. Como foi concluido na fase passada, a melhor combinação de valores para a constante cognitiva, social e da inércia é, respetivamente, 0.9, 0.9 e 0.2, sendo esses os valores que vamos utilizar para todas as fases de teste do algoritmo PSO.

Análise dos Resultados

Grid Search

Executando o algoritmo Grid Search, obtém-se os seguintes resultados:

GridSearchCV		
Parâmetros	Accuracy	Desvio Padrão
{'model__neurons': 256, 'optimizer__learning_rate': 0.0001}	0.866000	0.058172
{'model__neurons': 64, 'optimizer__learning_rate': 0.0001}	0.850000	0.046043
{'model__neurons': 64, 'optimizer__learning_rate': 0.001}	0.840000	0.051381
{'model__neurons': 64, 'optimizer__learning_rate': 0.01}	0.836000	0.057480
{'model__neurons': 128, 'optimizer__learning_rate': 0.0001}	0.838000	0.059800
{'model__neurons': 128, 'optimizer__learning_rate': 0.001}	0.840000	0.054772
{'model__neurons': 128, 'optimizer__learning_rate': 0.01}	0.842000	0.051536
{'model__neurons': 256, 'optimizer__learning_rate': 0.001}	0.848000	0.045782
{'model__neurons': 256, 'optimizer__learning_rate': 0.01}	0.844000	0.844000

Fig.13 – Tabela de resultados do Grid Search

Como se pode observar, a melhor configuração obtida corresponde a 256 neurónios e learning rate de 0.0001 com uma média de accuracy de 0.866 e desvio padrão de 0.058. Treinando o modelo com estas configurações no ficheiro *projeto.ipynb* obtém-se os seguintes resultados: Accuracy = 0.86, Precision = 0.859 e AUC = 0.9815.

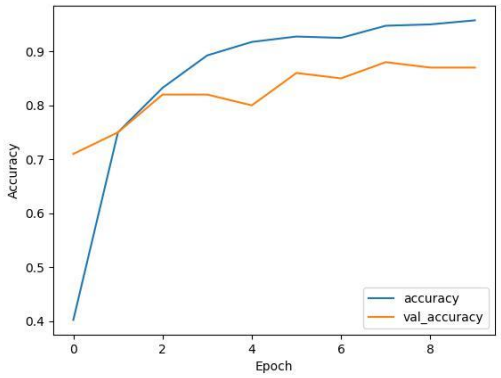

Fig.14 – Gráfico de accuracy de treino e validação do modelo de Grid Search

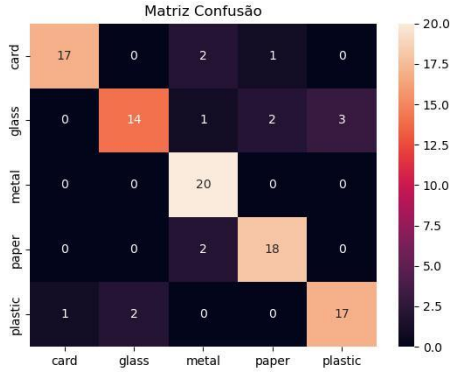

Fig.15 – Matriz Confusão do modelo de Grid Search

Através do gráfico da fig.14 é possível concluir que o modelo de Grid Search tem uma boa performance e não regista overfitting. A matriz confusão apresentada na fig.15, indica que o modelo tem alguma dificuldade a reconhecer imagens de vidro e que as confude mais com plástico.

Randomized Search

Executando o algoritmo Randomized Search, obtém-se os seguintes resultados:

RandomizedSearchCV		
Parâmetros	Accuracy	Desvio Padrão
{'optimizer__learning_rate': 0.0001, 'model__neurons': 128}	0.856000	0.055353
{'optimizer__learning_rate': 0.0001, 'model__neurons': 64}	0.834000	0.068877
{'optimizer__learning_rate': 0.001, 'model__neurons': 64}	0.830000	0.046476
{'optimizer__learning_rate': 0.01, 'model__neurons': 64}	0.842000	0.049558
{'optimizer__learning_rate': 0.001, 'model__neurons': 128}	0.842000	0.045343
{'optimizer__learning_rate': 0.01, 'model__neurons': 128}	0.848000	0.059127
{'optimizer__learning_rate': 0.0001, 'model__neurons': 256}	0.848000	0.053814
{'optimizer__learning_rate': 0.001, 'model__neurons': 256}	0.838000	0.054553
{'optimizer__learning_rate': 0.01, 'model__neurons': 256}	0.832000	0.042615

Fig.16 – Tabela de resultados do Randomized Search

Como se pode observar, ao contrário dos resultados obtidos pelo Grid Search, a melhor configuração obtida pelo Randomized Search corresponde a um learning rate de 0.0001 e 128 neurónios, apresentando uma média de accuracy de 0.856, inferior à obtida pelo Grid Search. Treinando o modelo com a configuração recebida, obtém-se os seguintes resultados: Accuracy = 0.86, Precision = 0.865 e AUC = 0.97475.

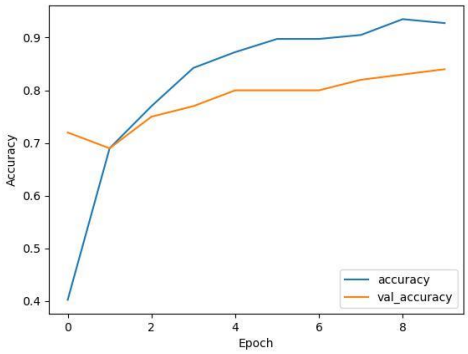

Fig.17 – Gráfico de accuracy de treino e validação do modelo de Randomized Search

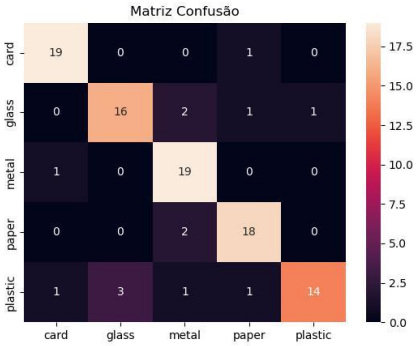

Fig.18 – Matriz Confusão do modelo de Randomized Search

Em termos de resultados, o modelo de Randomized Search apresenta a mesma accuracy que o modelo de Grid Search, melhor precision, porém um AUC pior. O gráfico da fig.17 apresenta uma boa performance do modelo, assim como a matriz de confusão que revela que o modelo desta vez tem dificuldade a reconhecer imagens de plástico sendo que as confunde com as de vidro.

PSO

Executando 8 vezes o algoritmo de PSO, obtém-se os seguintes resultados:

PSO - w=0.2, c1=0.9, c2=0.9		
Parâmetros	Valores	Accuracy
Neurónios: Learning Rate:	256 0.0076	0.81
Neurónios: Learning Rate:	253.822 0.0077	0.779
Neurónios: Learning Rate:	206.541 0.0001	0.86
Neurónios: Learning Rate:	208.525 0.00042	0.85
Neurónios: Learning Rate:	253.184 0.00016	0.89
Neurónios: Learning Rate:	196.178 0.0088	0.75
Neurónios: Learning Rate:	149.835 0.0034	0.829
Neurónios: Learning Rate:	212.6465 0.000285	0.86

Fig.19 – Tabela de resultados do PSO

Como se pode observar, a melhor configuração obtida utilizando o algoritmo PSO baseia-se em 253.184 neurónios e learning rate de 0.00016. Treinando o modelo com estas configurações no ficheiro *projeto.ipynb* obtém-se o seguinte resultado: Accuracy = 0.9, Precision = 0.898 e AUC = 0.9801.

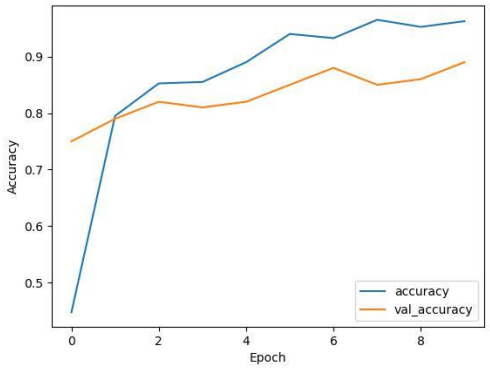

Fig.20 – Gráfico de accuracy de treino e validação do modelo de PSO

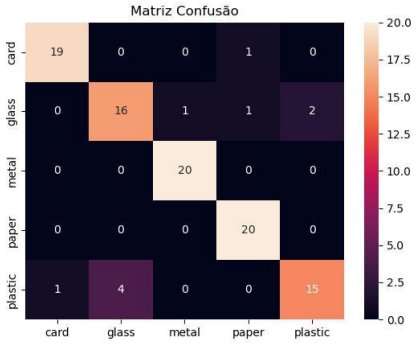

Fig.21 – Matriz Confusão do modelo de PSO

O modelo obtido pelas configurações do algoritmo PSO é o que apresentou melhores métricas de desempenho, chegando à accuracy de 0.9 e precision de 0.898. O gráfico da fig.20 revela o seu bom desempenho, assim como a matriz confusão que indica que o modelo conseguiu adivinhar a classe de todas as imagens de teste de metal, papel, faltando uma de cartão, porém apresenta os mesmos problemas que os outros modelos em reconhecer imagens de plástico e vidro.

Variação do dataset

Ao longo deste projeto fomos utilizando modelos apenas com aprendizagem por transferência. Como foi explicado, com este tipo de modelos não é preciso utilizar o dataset completo, sendo que uma pequena partição do mesmo é mais do que suficiente para obter ótimos resultados. Treinando o modelo com a melhor configuração obtida pelo algoritmo PSO, utilizando o dataset completo obtém-se os seguintes resultados: Accuracy = 0.9075, Precision = 0.907 e AUC = 0.9857

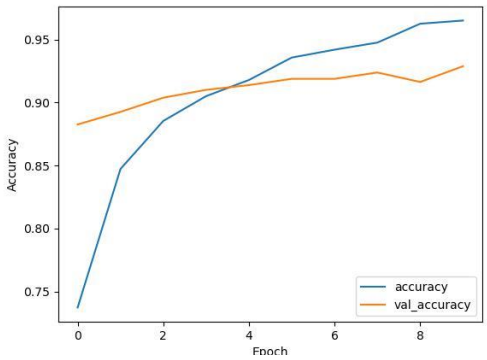

Fig. 22 – Gráfico de accuracy de treino e validação do melhor modelo usando o dataset completo

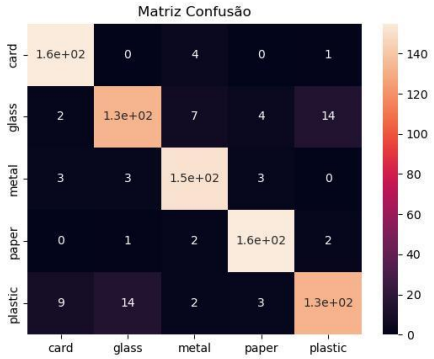

Fig. 23 – Matriz Confusão do melhor modelo utilizando o dataset completo

Utilizando o dataset completo com o melhor modelo, regista-se uma muito pequena melhoria nos valores da accuracy, precision e AUC, o que não justifica o maior custo computacional associado. Com apenas 12.5% do dataset original é possível obter resultados idênticos e o custo computacional associado é muito reduzido.

Comparação com fases anteriores

Nas fases anteriores, não conseguimos implementar uma rede MLP otimizada para o problema, portanto vamos utilizar uma rede MLP básica para a analisar e compará-la com os resultados obtidos com os modelos de aprendizagem por transferência. A rede MLP básica que utilizaremos é constituída por 300 neurónios na camada oculta, tem um número máximo de 100 iterações, random state com valor de 42 e o resto dos parâmetros com valor default.

```
mlp_model = MLPClassifier(hidden_layer_sizes=(300,), max_iter=100, random_state=42)
```

Fig.24 – Rede MLP básica utilizada para as comparações

Treinando esta rede MLP, obteve-se o seguinte desempenho: Accuracy = 0.72375, Precision = 0.73119 e AUC = 0.904.

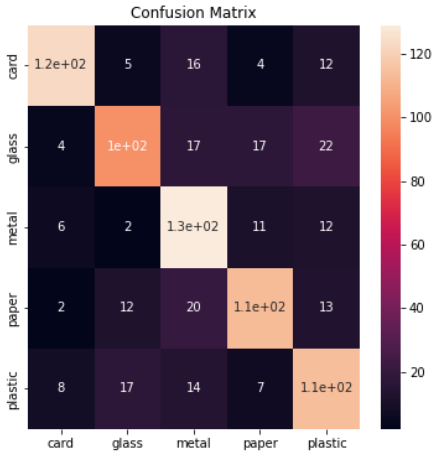

Fig. 25 – Matriz Confusão da Rede MLP

Utilizando o dataset completo, esta Rede MLP apresentou resultados consideravelmente inferiores aos apresentados pelos modelos com aprendizagem por transferência, podendo se concluir que modelos de aprendizagem por transferência apresentam melhores desempenhos, com custo computacional consideravelmente inferior comparado com Redes MLP.

Conclusões

Neste projeto foi possível aprender diversas coisas, desde o funcionamento dos modelos de aprendizagem por transferência, às camadas de dropout e outras técnicas de otimização de hiper-parâmetros para além dos algoritmos swarm.

As principais conclusões que retiramos são:

- Modelos de aprendizagem por transferência são superiores em todos os aspetos em relação às simples redes MLP, tanto a nível de desempenho, como de custo computacional.
- A camada de dropout é essencial para evitar situações de overfitting neste tipo de modelos.
- Grid Search e Randomized Search são ambas boas técnicas de otimização de hiper-parâmetros, tendo o Grid Search vantagem na procura pela melhor configuração de hiper-parâmetros.
- O algoritmo PSO é capaz de encontrar soluções ótimas para este tipo de problemas, sendo que foi o algoritmo de otimização que encontrou a melhor configuração de hiper-parâmetros.

Bibliografia

How to Grid Search Hyperparameters for Deep Learning Models in Python with Keras - MachineLearningMastery.com (utilizado no otimizacao.ipynb)

OpenAI. (2023). *ChatGPT* [Large language model]. https://chat.openai.com (utilizado para métricas de desempenho e no swarm.ipynb)

Intel Image Classification (CNN - Keras) | Kaggle (utilizada função para carregar dataset)

Garbage classification dataset (kaggle.com) (dataset)

What is the Dropout Layer? | Data Basecamp