Relatório Final

Redes Neuronais para a identificação de Pulsares

Mestrado Integrado em Engenharia Informática e Computação

IART - Inteligência Artificial

Turma 3MIEIC02, Grupo E1_3 :
André Miguel Ferreira da Cruz - 201503776
Edgar Filipe Amorim Gomes Carneiro - 201503784
João Filipe Lopes de Carvalho - 201504875

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, s/n, 4200-465 Porto, Portugal

20 de Maio de 2018

Conteúdo

1 Objetivo **2**

2 Especificação **2**
2.1 Descrição e análise do dataset. 2
2.2 Pré-processamento dos dados. 4
2.3 Modelo de aprendizagem a aplicar: redes neuronais. 4
2.4 Arquitetura das redes neuronais. 5
2.5 Configuração da rede neuronal. 6
2.6 Forma de Avaliação 6

3 Desenvolvimento **7**
3.1 Ferramentas utilizadas 7
3.2 Estrutura da aplicação 7
3.3 Detalhes de Implementação 9

4 Experiências **9**
4.1 Tamanho das *Hidden-Layers* 10
4.2 Algoritmos de Otimização de *Gradient Descent* 12
4.3 Reamostragem dos Dados da Classe Maioritária 13

5 Conclusões **15**

6 Melhoramentos **15**

7 Recursos **16**
7.1 Bibliografia 16
7.2 Software 16
7.3 Divisão do Trabalho pelos Elementos do Grupo 16

A Appendix **17**
A.1 Manual do Utilizador 17
A.2 Dependências das Bibliotecas 17

1 Objetivo

Este trabalho tem como objetivo a aplicação de Redes Neuronais artificiais na identificação de Pulsares. Neste sentido, começamos por expor e explicitar o tema, com uma pequena introdução à tarefa em questão, e às dificuldades à sua resolução (Secção 1). De seguida, analisamos o *dataset* em uso, descrevendo características e possíveis tendências (*bias*) indesejadas (Secção 2.1). No mesmo sentido, explicamos o pré-processamento necessário (Secção 2.2), bem como o modelo de aprendizagem a aplicar (Secção 2.3), a sua arquitetura (Secção 2.4), e as métricas usadas para avaliar adequadamente o desempenho do sistema (Secção 2.6). De seguida, é detalhado o ambiente de desenvolvimento usado, as ferramentas e bibliotecas a que recorremos, bem como a estrutura da aplicação desenvolvida (Secção 3). Finalmente, expomos todo o trabalho efetuado para resolver o problema, e avaliamos minuciosamente os modelos desenvolvidos e os seus resultados (Secção 4). Para concluir, enumeramos possíveis melhoramentos para trabalho futuro (Secção 6), e extraímos conclusões das experiências efetuadas e dos resultados obtidos (Secção 5).

Os pulsares são um tipo de estrelas de neutrões raro que, devido ao seu intenso campo magnético, transforma energia cinética (rotacional) em energia eletromagnética. Além disso, o seu campo magnético é suficientemente forte para arrancar partículas da sua superfície (na sua maioria eletrões), que são depois aceleradas na *magnetosfera* e emitidas sob a forma de um feixe estreito e contínuo. Este feixe de radiação, juntamente com a contínua rotação do pulsar, gera um sinal característico, análogo ao de um farol em rotação, de intensidade suficiente para ser detetado a milhões de anos-luz de distância. Desta forma, a deteção de um pulsar corresponde à procura e reconhecimento dos seus sinais periódicos.

Cada pulsar produz um padrão próprio, distinto dos restantes corpos celestes, e que varia ligeiramente entre diferentes pulsares. Assim, podemos detetar um sinal, realizando uma análise de alguns parâmetros, como por exemplo o tempo médio que um pulsar demora a reemitir o seu feixe para a terra. À primeira vista, concluir se um sinal é realmente proveniente de um pulsar pode parecer pouco complexo. No entanto, devido a fortes interferências de frequências de rádio e de sinais ruidosos que dificultam a receção de sinais genuínos, a grande maioria das supostas deteções são falsas.

Devido às dificuldades referidas anteriormente, e ao grande volume de dados em causa (induzido maioritariamente por falsas deteções), esta tarefa torna-se extenuante e dispendiosa para humanos, tornando-a uma muito boa candidata para a aplicação de técnicas de *Machine Learning*. Os algoritmos de *machine learning* funcionam através da construção de um modelo a partir de inputs amostrais, com a finalidade de fazer previsões ou decisões orientadas pelos dados e respetivas estatísticas, ao invés de seguir estruturas de decisão estáticas.

Assim sendo, neste trabalho iremos usar algoritmos de aprendizagem supervisionada, nomeadamente redes neuronais artificias, como forma de resolução do problema de classificação apresentado.

2 Especificação

Esta secção comporta a descrição dos dados usados, a explicitação da etapa de pré-processamento dos dados, bem como uma explicação mais detalhada do modelo de aprendizagem a aplicar. Por fim, é descrito o trabalho efetuado até ao momento, os resultados obtidos, as métricas usadas para a avaliação, bem como futuro trabalho no sentido de melhorar o desempenho do modelo.

2.1 Descrição e análise do dataset.

Nesta secção, é feita uma análise do *dataset* em uso, especificando o seu conteúdo e determinando as suas características e possíveis tendências (*bias*) indesejadas.

Na base de dados fornecida, cada candidato é caraterizado por oito variáveis continuas e uma classe. As oito variáveis contínuas distinguem-se em dois grupos, sendo as primeiras quatro referentes a estatísticas relativas ao *folded profile* de um pulsar, e as restantes quatro relativas às características da

curva *DM-SNR* (*Delta Modulation with Signal-to-Noise Ratio*). O primeiro grupo consiste num *array* de variáveis contínuas que descrevem uma versão pós-análise do pulsar. E as variáveis do segundo grupo são obtidas através da análise da curva *DM-SNR*. Esta consiste na modulação de um sinal analógico para digital ou o seu reverso. Por sua vez, a classe fornecida como último parâmetro tem um binário, indicando se o candidato era de facto um pulsar ou não. Em relação a cada um dos grupos são obtidas quatro variáveis contínuas: valor médio, desvio padrão, curtose e assimetria.

A base de dados fornecida contém um total de 17,898 entradas, sendo que destas, 16,259 tratam-se de dados espúrios causados pelo ruído (classe negativa) e as restantes 1,639 tratam-se de dados relativos a sinais de pulsares reais (classe positiva). Para melhor visualização dos dados fornecidos, a Figura 1 apresenta as distribuições das 8 variáveis contínuas de *input*. Essas mesmas distribuições encontram-se pormenorizadas na Figura 2, sob a forma de tabela.

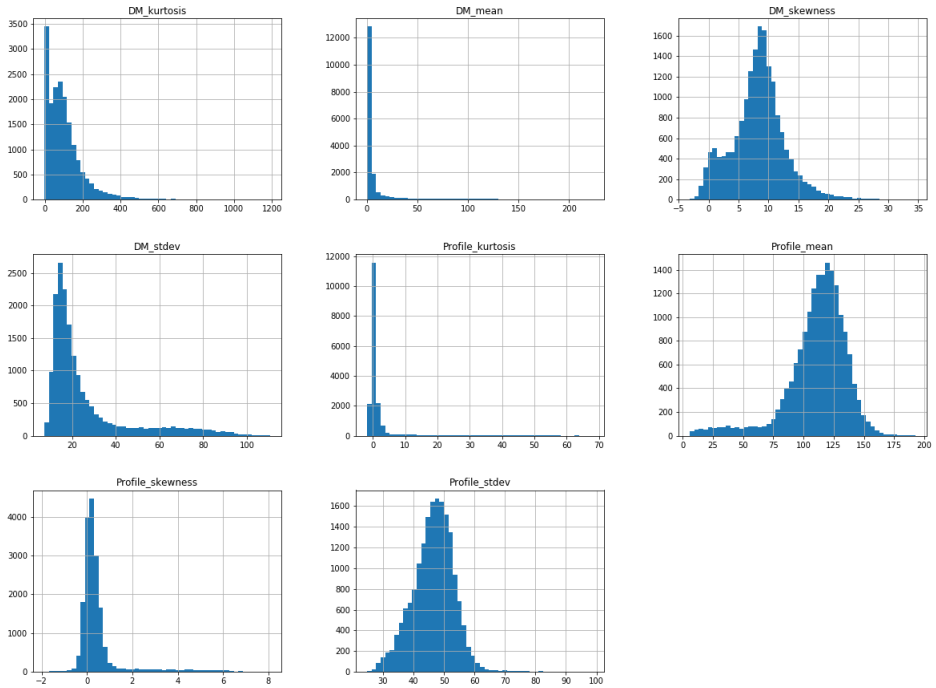

Figura 1: Distribuições relativas às 8 variáveis contínuas. Por ordem, da esquerda para a direita, e de cima para baixo, temos: Curtose da curva *DM-SNR*, Média da curva *DM-SNR*, Assimetria da curva *DM-SNR*, Desvio Padrão da curva *DM-SNR*, Curtose do *folded profile*, Média do *folded profile*, Assimetria do *folded profile* e Desvio padrão do *folded profile*.

Assim sendo, para esta tarefa de classificação binária os valores das oito variáveis relativas ao sinal são usados na análise de cada candidato, de forma a determinar se se trata ou não de um pulsar (informação fornecida na 9ª coluna do *dataset*).

	Profile_mean	Profile_stdev	Profile_skewness	Profile_kurtosis	DM_mean	DM_stdev	DM_skewness	DM_kurtosis
count	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000	17898.000000
mean	111.079968	46.549532	0.477857	1.770279	12.614400	26.326515	8.303556	104.857709
std	25.652935	6.843189	1.064040	6.167913	29.472897	19.470572	4.506092	106.514540
min	5.812500	24.772042	-1.876011	-1.791886	0.213211	7.370432	-3.139270	-1.976976
25%	100.929688	42.376018	0.027098	-0.188572	1.923077	14.437332	5.781506	34.960504
50%	115.078125	46.947479	0.223240	0.198710	2.801839	18.461316	8.433515	83.064556
75%	127.085938	51.023202	0.473325	0.927783	5.464256	28.428104	10.702959	139.309330
max	192.617188	98.778911	8.069522	68.101622	223.392141	110.642211	34.539844	1191.000837

Figura 2: Estatísticas descritivas sumárias, relativas às 8 variáveis de *input*.

2.2 Pré-processamento dos dados.

Nesta secção, é descrito qual o pré-processamento aplicado ao conjunto de dados, explicando também as vantagens que daí advêm.

Relativamente à etapa de pré-processamento dos dados, temos duas considerações importantes: por um lado, todas as colunas de *input* contêm valores de vírgula flutuante, de distribuição contínua; por outro lado, a coluna de *output*, corresponde a uma de duas classes (classe positiva ou negativa, caso seja ou não um pulsar).

Assim, para facilitar a aprendizagem por parte do algoritmo e contribuir para convergir mais rapidamente, fazemos normalização dos dados de input (*feature scaling*), para uma escala no alcance [-1,1]. Por sua vez, relativamente à classe de *output*, extraímos os dados para uma matriz unidimensional em que cada elemento tem um valor binário: 1 caso seja referente a um pulsar, e 0 em caso contrário.

2.3 Modelo de aprendizagem a aplicar: redes neuronais.

Neste trabalho iremos usar uma rede neuronal artificial, *Multilayer Perceptron* (MLP), que segue uma técnica de *aprendizagem supervisionada* denominada *backpropagation*. Assim, torna-se importante esclarecer cada um destes conceitos.

Algoritmos que seguem o modelo da aprendizagem supervisionada são algoritmos nos quais os possíveis *outputs* do algoritmo já são conhecidos e nos quais os dados usados para treinar o algoritmo já se encontram mapeados à resposta correta. Pode-se verificar que este modelo é aplicável ao problema dos pulsares na medida em que a totalidade das entradas na base de dados se encontram mapeados a um resultado, como na Secção 2.1.

Por sua vez, *backpropagation* é um método usado para o cálculo do gradiente de funções que regem as redes neuronais, sendo frequentemente usado no algoritmo de *gradient descent*. Este é um algoritmo iterativo de otimização de primeira ordem, que permite encontrar o mínimo local de uma função.

Dada uma rede neuronal artificial e uma função de erro (ou função de perda), o algoritmo de *backpropagation*, também conhecido por algoritmo da propagação de erros para trás (*backward propagation of errors*), calcula o gradiente da função de erro considerando sempre o peso das arestas na rede neuronal. Este cálculo é propagado para trás na rede neuronal, com o gradiente da última camada de pesos a ser calculado primeiro e o gradiente da primeira camada de pesos em último lugar (daí o seu nome *backpropagation*). Este fluxo invertido da informação de erro permite uma computação eficiente do gradiente em cada camada, ao invés da abordagem ingénua de calcular o gradiente de cada camada separadamente.

O algoritmo desenvolve-se em duas fases. Numa primeira fase, propaga-se através da rede de forma a chegar aos valores de *output*. De seguida, calcula o custo da propagação para cada output, sendo que esta estará intrinsecamente relacionada com a função de erro usada no algoritmo do gradiente descendente. Por fim, propaga-se em sentido contrário de forma a gerar um valor de erro para cada um dos neurónios.

Na segunda fase, os pesos das arestas são constantemente atualizados conforme o resultado desempenhado por cada um dos candidatos. Na atualização das arestas é usado uma percentagem previamente definida como fator de atualização dos pesos. Quanto mais elevado for este fator, mais rápido será o treino da rede neural. Por outro lado, quanto mais baixo for este fator, mais preciso será o treino.

2.4 Arquitetura das redes neuronais.

As redes neuronais, surgiram com base na perceção biológica das interligações presentes no cérebro humano. Assim, tal como o cérebro humano é constituído por neurónios, os quais recebem impulsos nervosos como entrada e saída de dados, também as redes neuronais são constituídas por camadas de neurónios, ativados com estímulos específicos. Um neurónio tem a capacidade de estimular os seus neurónios vizinhos, quando este é propriamente estimulado (função de ativação). Posto isto, podemos definir matematicamente um neurónio como sendo:

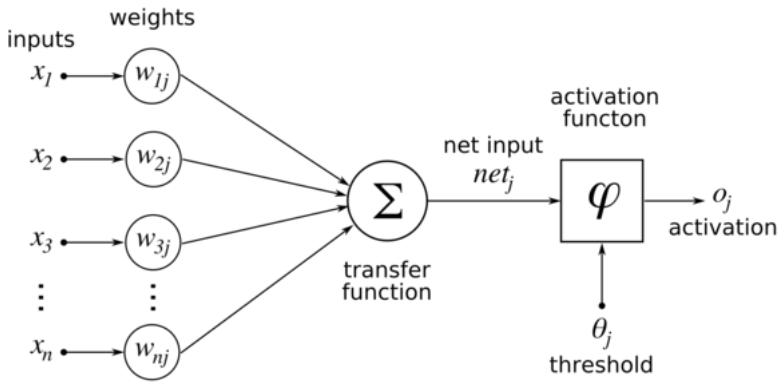

Figura 3: Modelo natemático de um neurónio.

$$y = f_{ativação}(\sum_i w_i x_i - \theta)$$

Figura 4: Representação matemática da saída de um neurónio.

Como o nome indica, é uma rede, por isso a sua consistência detêm vários neurónios conectados na forma de um grafo acíclico. As redes neuronais mais comuns, *feedforward*, são redes em que o *output* dos neurónios de uma camada é o *input* dos neurónios da camada seguinte.

Desta forma é possível separar a rede em diferentes camadas:

1. Uma ***Input Layer***, primeira camada na rede neuronal, responsável por receber os dados, determinar o tamanho do *dataset* e apresentar os padrões de reconhecimento. São apenas constituídas por neurónios de *input*.

2. Uma ou mais ***Hidden Layer***, é nesta(s) camada(s) que é feita maior parte da aprendizagem dos pesos do modelo. Camadas seguidas deverão corresponder a *features* cada vez mais abstratas, permitindo alcançar um modelo generalizável e de bom desempenho.

3. Uma ***Output Layer***, última camada da rede, responsável por apresentar os dados, esta camada é constituída com um número de neurónios geralmente igual ao número de classes de *output* (ou

apenas 1 se se tratar de classificação binária). É uma camada que não dispõe de função de ativação, e os cujos são exclusivamente neurónios de *output*.

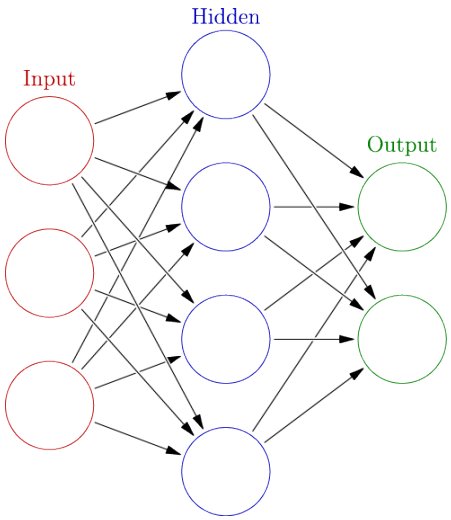

Figura 5: Representação Abstrata da Rede Neural

2.5 Configuração da rede neuronal.

Partindo agora para o nosso caso em concreto, é sempre muito complicado prever a configuração/arquitetura da rede ideal para um dado problema, isto é, qual o número ideal de *hidden layers*, qual o número de neurónios das camadas, qual o tipo de funções de ativação usadas, entre outras variações. Por isso é algo que requer alguns testes com diferentes composições para se obter os melhores resultados, coadjuvados por métodos de *Grid Search*, testando um conjunto de combinações promissoras e melhorando iterativamente.

Neste sentido, foram desenvolvidos testes ao número de *hidden layers*, variando entre 1 a 2 o número destas *layers*, e variando também o número de neurónios em cada uma entre 8, 12, 16, ou 32. Como função de ativação, foi usada a função "relu" para as camadas intermédias (*hidden*). A camada de *input* terá obrigatoriamente 8 entradas (uma por variável), e a camada de *output* tem apenas 1 neurónio, ativado pela função "sigmoid", de modo a balisar o *output* entre 0 e 1 (indicativo do grau de certeza de o candidato se tratar ou não de um pulsar). Os resultados destas configurações são apresentados na Secção 4.

2.6 Forma de Avaliação

Para nos assegurarmos que o modelo cumpre os requisitos, todos os dados reportados são resultantes de *k-fold cross-validation* (validação cruzada em k partes). Assim, numa fase inicial usamos *cross-validation* com $k = 3$ (para um treino mais rápido dos modelos, e assim uma pesquisa mais rápida sobre os possíveis hiperparâmetros), e, após terem sido identificados alguns modelos promissores, prosseguimos com $k = 5$.

Adicionalmente, e tendo em conta a tarefa em questão, será muito útil ter em atenção o *recall* da classe positiva, pois queremos maximizar a identificação de pulsares. Neste sentido, as métricas usadas para avaliação dos modelos serão numa primeira fase a exatidão global do modelo (*accuracy*), e numa segunda fase uma avaliação mais detalhada com o *recall* da classe positiva, e o *F1 score*. O *F1 score* consiste numa média harmónica da precisão e do *recall* para cada uma das classes em causa.

3 Desenvolvimento

Nesta secção, será analisado o desenvolvimento do trabalho em si, nomeadamente quais as ferramentas e linguagens de programação utilizadas, qual a estrutura da aplicação concebida e os detalhes da sua implementação.

3.1 Ferramentas utilizadas

Para o desenvolvimento da aplicação foi usada a linguagem de programação *Python3*.

Foram usadas várias bibliotecas da linguagem referida, sendo de importante menção as seguintes: *keras*, para modelação rápida de alto nível da rede neuronal; *Tensorflow*, para modelação de baixo nível da rede neuronal, proporcionando maior liberdade em detrimento do tempo de desenvolvimento; *scikit-learn*, para análise dos modelos gerados, realização de validação cruzada e cálculo de métricas globais como *F1-score*; *imblearn*, útil no manuseamento de conjuntos de dados desequilibrados, possibilitando fácil reamostragem dos dados usados na aplicação, nomeadamente *undersampling*; *numpy*, para execução de operações sobre matrizes com elevada *performance*; *matplotlib* para criação de diversos gráficos (*e.g.* Figura 1); finalizando, *tensorboard*, para visualização de métricas e do seu progresso no decorrer do treino, bem como visualização das ativações de neurónios da rede neuronal.

Outras bibliotecas usadas, mas de menor relevância, são ainda: *absl-py*, *astor*, *bleach*, *gast*, *grpcio*, *h5py*, *html5lib*, *pandas*, *protobuf*, *python-dateutil*, *pytz*, *PyYAML*, *scipy*, *six*, *termcolor* e *Werkzeug*.

A aplicação foi ainda desenvolvida e testada em três sistemas operativos, nomeadamente, Linux, MacOS e Windows.

3.2 Estrutura da aplicação

É possível dividir o trabalho em cinco fases distintas:

1. Carregamento dos dados e repartição dos mesmos em dados de treino e de teste, de forma a melhor controlar o fenómeno de *overfitting*.
2. Aplicação opcional de *undersampling*.
3. Criação do modelo da rede neuronal.
4. Treino da rede neuronal.
5. Avaliação da rede neuronal.
6. Geração de gráficos referentes ao treino e teste da rede neuronal.

Através da análise do ficheiro *main.py*, é possível distinguir as respetivas chamadas as quatro fases referidas.

Assim, para a primeira fase referida - carregamento dos dados e repartição dos mesmos -, a respetiva secção de código é:

```
1 pulsars = load_pulsar_csv()
2 train_set, test_set = train_test_split(pulsars, 0.2)
3
4 X_train, Y_train = train_set[:, :-1], train_set[:, -1]
5 X_test, Y_test = test_set[:, :-1], test_set[:, -1]
```

Neste excerto de código, começamos por guardar na variável *pulsar* os dados que nos são fornecidos. de seguida dividimos esses dados, em dois conjuntos, os dados usados para testar a rede neuronal e os dados usados para avaliação da rede neuronal. Nas duas linhas seguintes, definimos o *input* e o *output* para cada um dos conjuntos previamente criados.

Para a segunda fase referida - Aplicação de *undersampling* -, a respetiva secção de código é:

```
from imblearn.under_sampling import RandomUnderSampler
rus = RandomUnderSampler(return_indices=True)
X_resampled, Y_resampled, idx_resampled = rus.fit_sample(pulsars[:,:-1], pulsars[:,-1])
print("Original data size: %d. Undersampled data size: %d" % (len(pulsars), len(
    X_resampled)))

X_train, Y_train = X_resampled, Y_resampled
```

Antes de procedermos à explicação do excerto de código apresentado, consideramos necessário deixar transparente o que é *undersampling*. Por *undersampling* entende-se a eliminação aleatória de dados da classe maioritária, de forma a reduzir o impacto desta sobre o conjunto de dados.

Neste excerto de código, são criadas duas variáveis de *input* e *output* resultantes da aplicação de *undersampling* ao conjunto de dados, sendo estas posteriormente usadas como *input* e *output* do conjunto de treino.

Para a terceira fase referida - criação do modelo da rede neuronal -, a respetiva secção de código é:

```
model = create_model(np.size(X_train, axis=1))
```

A função *create_model* encontra-se definida no ficheiro *modes.py*, sendo a sua definição:

```
def create_model(input_dim):
    input = Input(shape=(input_dim,))
    x = Dense(32)(input)
    x = LeakyReLU(alpha=0.3)(x)
    x = Dense(16)(x)
    x = Activation('relu')(x)
    output = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=input, outputs=output)
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Neste excerto de código, é definido o modelo a ser utilizado pela rede neuronal. Este é constituído por duas camadas, tendo a primeira um total de trinta e dois neurónios e a segunda dezasseis neurónios. Na secção 4 será feita uma análise minuciosa sobre o modelo usado.

Para a quarta fase referida - treino da rede neuronal -, a respetiva secção de código é:

```
## Optionally: Balance Class Weights
from sklearn.utils import class_weight
class_weight = class_weight.compute_class_weight('balanced',
                                                 np.unique(Y_train),
                                                 Y_train)
print("Class weight: ", class_weight)
#

# Train Model
model.fit(X_train, Y_train, epochs=100, batch_size=16,
          validation_data=(X_test, Y_test),
          callbacks=callbacks,
          class_weight={0: class_weight[0], 1: class_weight[1]}
          )
```

Neste excerto de código, o modelo criado é treinado usando o conjuntos de dados de treino previamente computados. Estes encontram-se guardados nas variáveis *X_train* e *Y_train*. De notar que é efetuado um estudo em relação ao peso das classes, sendo este detalhadamente explicado na secção 4.

Para a quinta fase referida - avaliação da rede neuronal -, a respetiva secção de código é:

```
scores = model.evaluate(X_test, Y_test)
print("Overall Accuracy: %.2f\n" % (scores[1] * 100))
print(evaluate_classwise(model, X_test, Y_test))
```

Neste excerto de código, é feita a avaliação do modelo usando o conjunto de dados previamente guardados nas variáveis *X_test* e *Y_test*. Posteriormente, é também avaliado o desempenho do modelo em cada classe (precisão, *recall*, *F1-score*).

3.3 Detalhes de Implementação

Para além do uso normal da plataforma *TensorBoard*, foi extendida com ajuda a software de código livre de modo a facilitar a criação de gráficos e outras visualizações.


```
scores = model.evaluate(X_test, Y_test)
print("Overall Accuracy: %.2f\n" % (scores[1] * 100))
print(evaluate_classwise(model, X_test, Y_test))
```


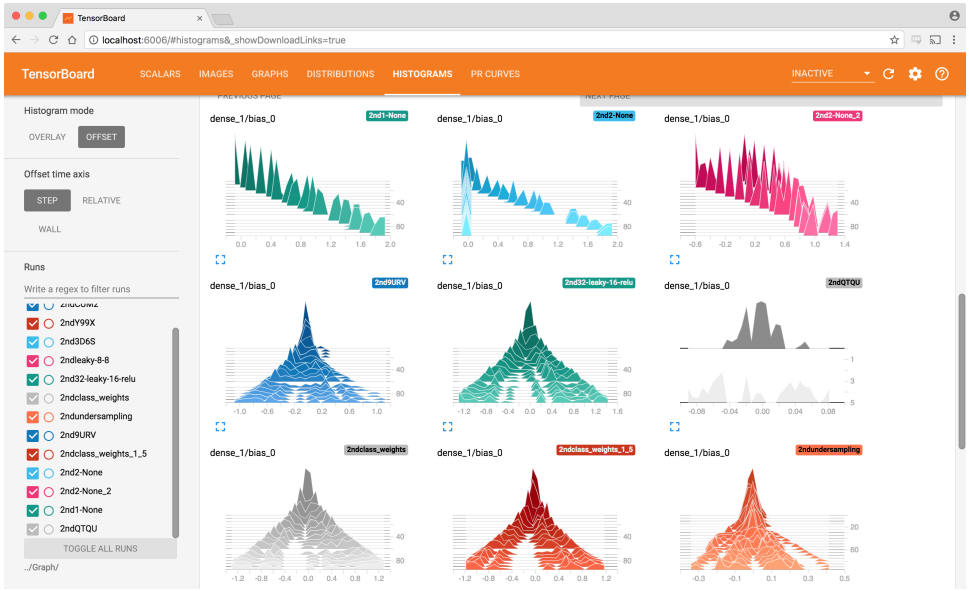

Figura 6: Exatidão dos modelos em função do número de *epochs* de treino, medida num dataset reservado para validação. Legenda: a cinzento a arquitetura 32-16; a verde a arquitetura 16-16; a azul escuro a arquitetura 8-16; a magenta a arquitetura 4-16; e a azul claro a arquitetura 2-16.

4 Experiências

Esta secção expõe detalhadamente todas as experiências feitas para melhor solucionar o problema de classificação de pulsares. Para isso, como visto nas secções anteriores, iremos utilizar redes neuronais artificiais *feedforward*. Neste sentido, as experiências incidem sobre o tamanho das *hidden-layers*, as suas funções de ativação, a função de treino utilizada, e, por último, técnicas para equilibrar as duas classes de um *dataset* desequilibrado. Para além disso, todas as redes geradas têm algumas características em comum:

- Uma camada de entrada, com 8 neurónios (1 por cada *feature* de que dispomos);
- Uma camada de saída, com 1 neurónio, ativada pela função *sigmoid*, de modo a que o output da rede seja a probabilidade de a instância em causa ser um pulsar;
- Uma ou duas camadas escondidas, de tamanho variável;
- A função de perda será *binary crossentropy*;
- A *seed* de aleatoriedade foi fixada no valor 42, para ser possível reproduzir facilmente os resultados;

4.1 Tamanho das *Hidden-Layers*

Esta experiência teve como objetivo determinar qual o melhor número de neurónios das camadas escondidas da rede. Neste sentido foi feita uma pesquisa exaustiva (*grid search*) sobre um conjunto de tamanhos que achamos promissores. Os resultados, obtidos ao fim de 100 *epochs* e validados por *3-fold cross-validation*, estão apresentados na Tabela 1.

First Hidden Layer	Second Hidden Layer	Accuracy
32	16	97.93
32	12	97.57
32	4	97.58
32	2	97.50
32	-	97.49
16	16	97.87
16	12	97.52
16	4	97.46
16	2	97.17
16	-	97.31
8	16	97.69
8	12	97.52
8	4	97.16
8	2	97.21
8	-	97.11
4	16	97.65
4	12	97.54
4	4	97.36
4	2	97.12
4	-	96.97
2	16	96.99
2	12	97.02
2	4	96.94
2	2	96.76
2	-	96.50

Tabela 1: Resultados obtidos com diferentes configurações da rede neuronal, por ordem decrescente da primeira coluna, e de seguida por ordem decrescente da segunda coluna.

Como podemos ver, atingimos resultados muito promissores, sendo que o melhor modelo obteve exatidão de 97.93%, e tem duas *hidden layers*, a primeira com 32 neurónios e a segunda com 16 neurónios. Como indicado no início desta secção, esta experiência é facilmente reproduzível, pois a *seed* de aleatoriedade foi fixada no valor 42.

Adicionalmente, é importante notar que a diferença entre os desempenhos dos diferentes modelos é diminuta, tendo a esmagadora maioria das configurações de rede resultado em valores de exatidão entre 97% e 98%. O desempenho de alguns modelos ao longo dos *epochs* de treino está disponível na figura 7, mostrando apenas as arquiteturas do tipo x-16 para não poluir o gráfico, mantendo os dados interpretáveis. A perda dos mesmos modelos ao longo dos *epochs* de treino está disponível na figura 8.

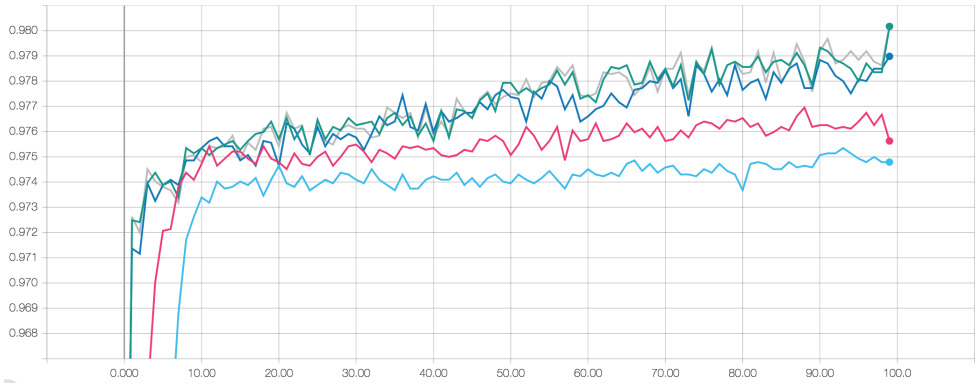

Figura 7: Exatidão dos modelos em função do número de *epochs* de treino, medida num dataset reservado para validação. Legenda: a cinzento a arquitetura 32-16; a verde a arquitetura 16-16; a azul escuro a arquitetura 8-16; a magenta a arquitetura 4-16; e a azul claro a arquitetura 2-16.

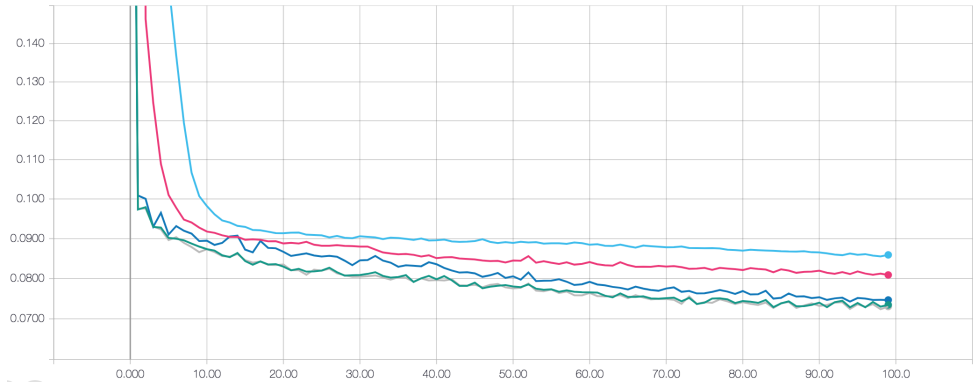

Figura 8: Perda dos modelos em função do número de *epochs* de treino, medida num dataset reservado para validação. Legenda: a cinzento a arquitetura 32-16; a verde a arquitetura 16-16; a azul escuro a arquitetura 8-16; a magenta a arquitetura 4-16; e a azul claro a arquitetura 2-16.

É também interessante notar que uma rede muito simples, com apenas uma *hidden layer* com dois neurónios, conseguiu um valor de exatidão de 96.50%, apenas 1.43% menos do que o melhor modelo. Achamos que isto se deve ao diminuto tamanho do *dataset* que usamos, não sendo suficiente para treinar modelos mais complexos, e portanto não aproveitando todo o potencial de que um modelo com duas *hidden layers* dispõe. Para além disso, também estará relacionado com a distribuição das classes no *dataset* (a classe negativa representa cerca de 91% dos dados), o que faz com que facilmente se atinjam os 91% de exatidão. No entanto, é importante notar que os modelos mais complexos não sofreram overfitting, alcançando os melhores valores de desempenho, e sendo estáveis ao longo da validação cruzada (*cross validation*).

4.2 Algoritmos de Otimização de *Gradient Descent*

Esta experiência teve como objetivo determinar qual o melhor algoritmo de otimização de *gradient descent* a usar, de entre elas: *RMSprop*, *Adam*, *AdaMax*, *Nadam*, *Adadelta*, ou simplesmente *Stochastic Gradient Descent*. Foram usados apenas algumas configurações de rede selecionadas da experiência anterior, de modo a controlar o tempo de computação (acabando por demorar cerca de 3 horas). Os resultados, obtidos ao fim de 100 *epochs* e validados por *3-fold cross-validation*, estão apresentados na Tabela 2.

Optimizer	First Hidden Layer	Second Hidden Layer	Accuracy
SGD	32	16	94.23
RMSprop	32	16	96.63
Adadelta	32	16	97.66
Adam	32	16	90.36
AdaMax	32	16	97.68
Nadam	32	16	90.52
SGD	32	8	97.43
RMSprop	32	8	97.76
Adadelta	32	8	97.72
Adam	32	8	93.21
AdaMax	32	8	97.75
Nadam	32	8	96.75
SGD	32	4	95.74
RMSprop	32	4	97.83
Adadelta	32	4	96.35
Adam	32	4	96.24
AdaMax	32	4	97.77
Nadam	32	4	97.94

Tabela 2: Valores de exatidão de modelos treinados com diferentes funções de otimização de treino, separados por diferentes configurações da rede neuronal.

Os resultados obtidos são algo surpreendentes, mostrando a instabilidade de treino de configurações de rede maiores: a configuração 32-16 obteve valores inconsistentes, enquanto a configuração 32-4 obteve consistentemente bons resultados. Esta inconsistência deve-se possivelmente à maior dificuldade por parte de maiores redes neuronais de escaparem a estados de equilíbrio locais.

O melhor resultado desta pesquisa exaustiva de hiperparâmetros foi obtido pelo conjunto (Nadam, 32, 4), ou seja, o otimizador *Nadam*, com uma rede com 32 neurónios na primeira camada escondida, e 4 neurónios na segunda camada escondida. Este valor é consideravelmente melhor que o previamente obtido para esta configuração ($97.94 > 97.58$). Este conjunto foi ainda testado mais exaustivamente, com os dados de precision, recall, e *f1* de cada classe disponíveis na tabela 3.

	Precision	*Recall*	*F1*
Classe Negativa	99	99	99
Classe Positiva	92	85	88

Tabela 3: Valores de *precision*, *recall* e *F1-score* para a configuração de rede 32-4, treinada com o otimizador *Nadam*.

4.3 Reamostragem dos Dados da Classe Maioritária

Uma das principais dificuldades deste problema é o facto de as frequências das duas classes serem extremamente desequilibradas (numa razão de 10 para 1). Para tentar colmatar esta dificuldade fizemos uso de duas técnicas: *resampling* dos dados da classe maioritária (a classe negativa), de modo a igualar o número de instâncias da classe minoritária (a classe positiva); e a alteração da perda no treino do modelo consoante a classe, de modo a que cada instância da classe minoritária conte como 5 instâncias da classe maioritária. Os resultados do primeiro teste (de *undersampling*) estão disponíveis na tabela 4, e os resultados o segundo teste (de diferentes pesos para cada classe) estão disponíveis na tabela 5.

	Precision	*Recall*	*F1*
Classe Negativa	89	96	92
Classe Positiva	95	87	91

Tabela 4: Valores obtidos num *dataset* reamostrado, com número igual de instâncias de cada classe. Valores referentes à melhor configuração de rede da experiência anterior.

	Precision	*Recall*	*F1*
Classe Negativa	99	98	99
Classe Positiva	84	92	88

Tabela 5: Valores obtidos com treino com peso das classes modificado: erros na classe positiva valem 5 vezes mais que erros na classe negativa. Valores referentes à melhor configuração de rede da experiência anterior.

Os resultados obtidos estão em linha com o esperado, havendo um aumento generalizado do desempenho na classe positiva, acompanhado com uma diminuição do desempenho na classe negativa. É de particular importância o facto de a realização de *undersampling* aumentar consideravelmente a precisão na classe positiva, de 92% para 95%, e diminui a precisão na classe negativa de 99% para 89%. É também interessante o facto de a modificação dos pesos das classes ter aumentado substancialmente o *recall* da classe positiva, de 85% para 92%, mas piorado a precisão na mesma classe, de 92% para 84%, mantendo o score *F1* nos 88%.

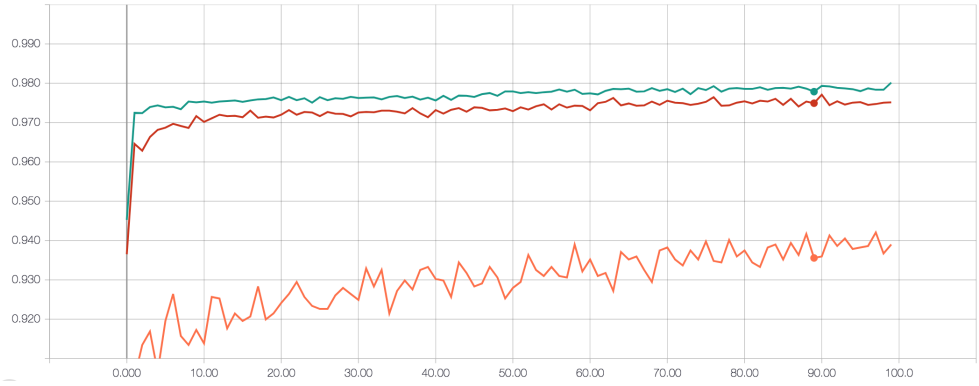

Figura 9: Exatidão dos modelos em função do número de *epochs* de treino, medida num dataset reservado para validação. Legenda: a verde o melhor modelo da experiência anterior (Secção 4.2); a vermelho a experiência resultante da alteração dos pesos das classes (Tabela 5); e a laranja a experiência resultante de *undersampling* (Tabela 4).

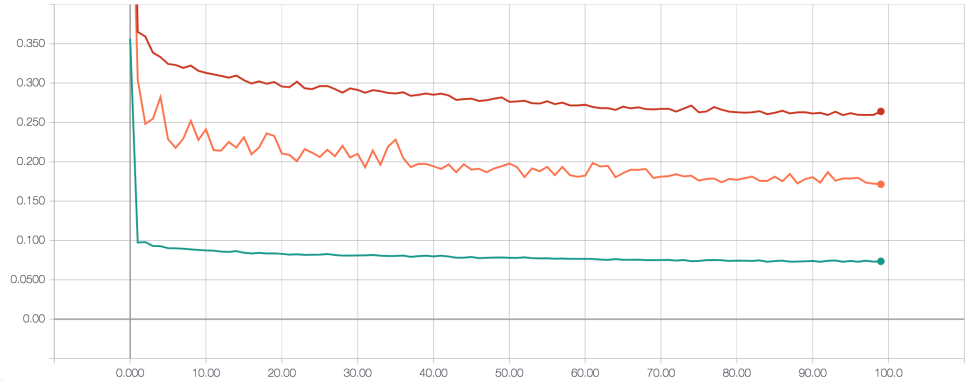

Figura 10: Perda dos modelos em função do número de *epochs* de treino, medida num dataset reservado para validação. Legenda: a verde o melhor modelo da experiência anterior (Secção 4.2); a vermelho a experiência resultante da alteração dos pesos das classes (Tabela 5); e a laranja a experiência resultante de *undersampling* (Tabela 4).

Adicionalmente, expomos os gráficos da precisão (Figura 9) e da perda (Figura 10) de ambos os modelos, bem como do melhor modelo da experiência anterior (Secção 4.2) para comparação. É particularmente interessante o facto de o modelo que foi treinado com pesos de perda alterados nas diferentes classes ter melhor resultados que o modelo treinado nos dados *undersampled*, no entanto ter também maior perda do que este, resultante de a perda na classe positiva contar 5 vezes mais para o primeiro modelo.

Achamos também interessante apresentar a curva de precisão da classe positiva em função do *recall* da mesma classe, consoante o limiar de separação das duas classes é alterado, e que normalmente é 0.5 (Figura 11). É evidente a separação do modelo resultante de dados *undersampled* dos restantes modelos, conseguindo uma melhor precisão a níveis de *recall* próximos do máximo. Tendo isto em conta, podemos concluir que se o objetivo for a captação do maior número possível de instâncias de Pulsar, este será o modelo mais indicado, obrigando no entanto a uma posterior triagem de falsos positivos.

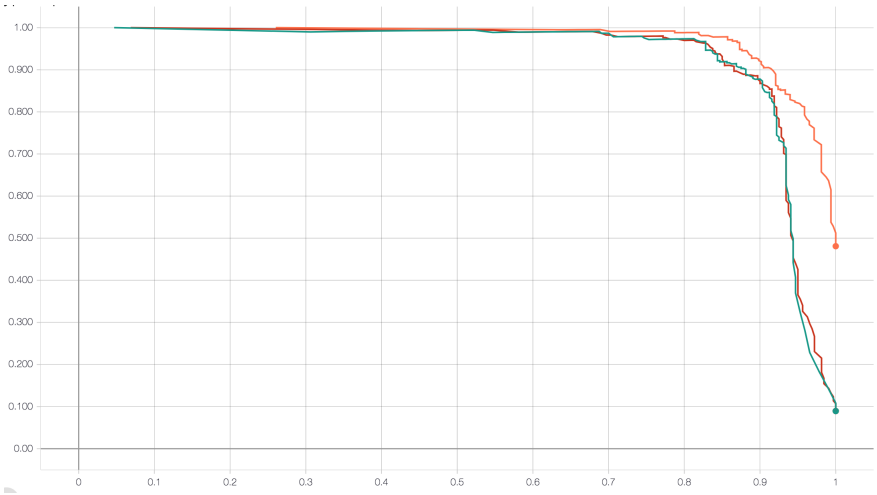

Figura 11: Precisão da classe positiva em função do *Recall* da mesma classe, consoante o limiar de classificação das duas classes é alterado. Legenda: a verde o melhor modelo da experiência anterior (Secção 4.2); a vermelho a experiência resultante da alteração dos pesos das classes (Tabela 5); e a laranja a experiência resultante de *undersampling* (Tabela 4).

5 Conclusões

Primeiramente, acreditamos que o nosso conhecimento sobre redes neuronais foi consideravelmente aprofundado, mostrando-se assim o grupo satisfeito com a escolha de tema realizada. De facto, Redes Neuronais artificiais representam uma abordagem a problemas bastante atuais, contribuindo assim ainda mais para o interesse do grupo no trabalho.

É também evidente que Redes Neuronais artificiais se revelam uma ferramenta extremamente útil, apropriada para resolver problemas semelhantes ao apresentado, resultando numa abordagem ao problema bastante diferente de tudo aquilo com o que nós, como alunos do 3º ano de Informática, tínhamos tido a oportunidade de trabalhar.

Inicialmente, os resultados obtidos revelaram-se promissores sendo a precisão do melhor modelo *97.54%*. No entanto, o grupo considerava ser possível a

Em suma, os resultados obtidos revelaram-se promissores, sendo a precisão do melhor modelo apresentado cerca de 98%. Ainda assim, o grupo considera ser possível melhorar o desempenho da rede neuronal artificial implementada, através da experimentação com diferentes camadas e configurações.

6 Melhoramentos

Durante todo o trabalho fizemos uma pesquisa exaustiva da configuração da rede e dos seus hiperparâmetros, o que nos leva a acreditar na qualidade dos nossos resultados.

Mesmo assim poderíamos ter alguns melhoramentos na pesquisa mais aprofundada de alguns hiperparâmetros como o *learning rate*, *Grid Search*, entre outros...

Para além disso podíamos basear num maior *dataset*, este sendo também mais equilibrado. Dado que o *dataset* usado no treino da nossa rede, tinha uma vasta discrepância na quantidade de pulsares válidas.

Com estes melhoramentos levaria-nos a uma melhoria garantida do desempenho dos modelos.

7 Recursos

7.1 Bibliografia

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin et al. "TensorFlow: A System for Large-Scale Machine Learning."In OSDI, vol. 16, pp. 265-283. 2016.

Chollet, François. "Keras."KerasCN Documentation (2017): 55.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. "Scikit-learn: Machine learning in Python."Journal of machine learning research 12, no. Oct (2011): 2825-2830. Harvard

Lemaître, Guillaume, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning."Journal of Machine Learning Research 18.17 (2017): 1-5.

Lyon, Robert James. "Why are pulsars hard to find?."PhD diss., University of Manchester, 2016.

R. J. Lyon, B. W. Stappers, S. Cooper, J. M. Brooke, J. D. Knowles, Fifty Years of Pulsar Candidate Selection: From simple filters to a new principled real-time classification approach, Monthly Notices of the Royal Astronomical Society 459 (1), 1104-1123, DOI: 10.1093/mnras/stw656

Ruck, Dennis W., Steven K. Rogers, Matthew Kabrisky, Mark E. Oxley, and Bruce W. Suter. "The multilayer perceptron as an approximation to a Bayes optimal discriminant function."IEEE Transactions on Neural Networks 1, no. 4 (1990): 296-298.

7.2 Software

- IDE: Visual Studio Code
- Linguagem de Programação: Python3
- Bibliotecas: Keras, Tensorflow, Scikit-learn, Imbalanced-learn
- Ferramentas: TensorBoard

7.3 Divisão do Trabalho pelos Elementos do Grupo

- André Miguel Ferreira da Cruz: 38%
- Edgar Filipe Amorim Gomes Carneiro: 32%
- João Filipe Lopes de Carvalho: 30%

A Appendix

A.1 Manual do Utilizador

Instalar dependências:

1. Abrir um novo terminal na raiz do projeto;
2. Instalar todas as dependências, como descrito a baixo, executando o comando:

```
pip3 install -r requirements.txt
```

Executar através do terminal:

1. Abrir um novo terminal na pasta "./src"do projeto;
2. Executar o seguinte comando para treinar e avaliar o modelo:

```
python3 main.py
```

3. Executar o seguinte comando para executar *GridSearch* sobre uma rede de hiperparâmetros (referente à experiência da Secção 4.1):

```
python3 grid_search.py
```

Executar com interface *Jupyter*:

1. Abrir um novo terminal na pasta do projeto;
2. Executar o comando:

```
jupyter notebook
```

3. Na *Dashboard* do Jupyter, selecionar o projeto pretendido dentro da pasta "./src/notebook";
4. Na nova janela aberta, carregar no botão *Run*;
5. Por fim, esperar que a simulação acabe para visualizar os resultados.

A.2 Dependências das Bibliotecas

Para correr o código fornecido em anexo é necessário instalar as devidas dependências e bibliotecas. Para além das bibliotecas e ferramentas enumeradas na Secção 7.2, é necessário instalar as dependências usadas indiretamente. Para tal, e supondo que tem o python3 e o pip3 instalados, deve correr o seguinte comando a partir da raiz do projeto:

```
pip3 install -r requirements.txt
```

Sendo o ficheiro *requirements.txt* fornecido juntamente com o código, e tendo o seguinte conteúdo:

```
absl-py==0.2.1
astor==0.6.2
bleach==1.5.0
gast==0.2.0
grpcio==1.12.0
h5py==2.7.1
html5lib==0.9999999
imbalanced-learn==0.3.3
imblearn==0.0
Keras==2.1.6
Markdown==2.6.11
numpy==1.14.3
protobuf==3.5.2.post1
PyYAML==3.12
scikit-learn==0.19.1
scipy==1.1.0
six==1.11.0
sklearn==0.0
tensorboard==1.8.0
tensorflow==1.5.0
tensorflow-tensorboard==1.5.1
termcolor==1.1.0
Werkzeug==0.14.1
```