Relatório Final

Inteligência Artificial
2017/2018

Mestrado Integrado em Engenharia Informática e Computação

Redes Neuronais para a Identificação de Pulsares

Afonso Jorge Ramos	up201506239@fe.up.pt
Bárbara Sofia Silva	up201505628@fe.up.pt
Julieta Pintado Jorge Frade	up201506530@fe.up.pt

20 de Maio 2018

Conteúdo

1 Objetivo

Este trabalho tem como finalidade a aplicação de redes neuronais artificiais na identificação de pulsares.

Os pulsares são estrelas de neutrões altamente magnetizadas que se formam quando uma estrela mais massiva que o Sol colapsa. O seu campo magnético é mais forte que o da Terra e a radiação é irradiada dos seus polos. De facto, é nos polos de um pulsar onde o campo magnético é mais intenso, resultando na produção de radiação em forma de ondas rádio, que por sua vez são emitidas num raio de radiação muito apertado, à semelhança de uma lanterna.

Adicionalmente, devido à sua rotação, cada pulsar produz um padrão de emissão diferente. Também, a existência de interferências rádio e *signal noise* prejudicam a sua deteção, o que faz com que encontrar um pulsar se torne bastante difícil. De modo a facilitar este processo, o uso das ferramentas de *Machine Learning* é fundamental, pois a classificação de pulsares candidatas é automatizada, obtendo assim uma rápida análise dos dados.

Machine Learning consiste na capacidade de uma máquina reconhecer padrões e na habilidade dos computadores aprenderem sem serem explicitamente programados. Além disso, foca-se em um dos dois raciocínios da inteligência artificial, o **indutivo**, visto que extrai regras e padrões de grandes conjuntos de dados. Neste caso, o objetivo é treinar o computador para que este seja capaz de identificar um pulsar.

2 Especificação

2.1 Análise detalhada do tema

À semelhança do que foi explicado na secção anterior, o tema proposto tem como finalidade a aplicação de redes neuronais na identificação de pulsares.

Na verdade, um pulsar é um relógio muito preciso. O seu *tick* é o breve pulso de ondas de rádio, e o intervalo regular é o período do pulsar, que é o tempo que leva para o pulsar girar uma vez. Após a medição de vários pulsos, concluímos que os pulsares têm períodos extremamente estáveis e, como tal, podemos prever com uma precisão muito alta quando qualquer pulso deve chegar à Terra. Porém, inúmeros fatores podem fazer com que o tempo de chegada de um determinado pulso seja diferente do previsto. Os astrónomos podem modelar todos estes fatores e obter uma previsão coerente.

Certamente, o grande desafio em usar os pulsares na ciência é identifica-los, dado que a quantidade de dados a analisar é muito grande. Por esta razão, é necessário implementar uma forma automática de identificação, que é onde as ferramentas de *Machine Learning* entram.

2.2 Ilustração de cenários

Os astrónomos usam pulsares como ferramentas para estudar o universo. Visto que é impossível criar algo semelhante a um pulsar na Terra, essas ferramentas são únicas e permitem responder a perguntas exclusivas. Assim, concluímos que os pulsares têm uma enorme importância na astronomia.

2.3 Explicação de Dataset

O *dataset* que nos foi providenciado provém de uma amostra de pulsares candidatos recolhida durante a *High Time Resolution Universe Survey, South (HTRU2)*. Um Pulsar é uma estrela de neutrões relativamente rara, que produz emissões de rádio detetáveis no nosso planeta, pelo que, são de elevado interesse científico, tanto como sondas do espaço-tempo e meio interestelar, bem como estado da matéria.

Com a rotação dos pulsares, chega-nos um padrão de emissões de rádio *broadband*, padrão esse que se repete conforme a velocidade de rotação dos pulsares, sendo que, a deteção destes padrões de ondas de rádio, nos ajudam a identificar, com a maior certeza possível, estes pulsares. No entanto, cada pulsar produz um padrão de emissão

ligeiramente diferente, que varia com a rotação, logo é necessário calcular a média do padrão ao longo das várias rotações. Para além disso, a maioria das deteções de pulsares são falsos positivos causados por interferências da frequência de rádio (RFI) e *signal noise*, o que torna sinais legítimos difíceis de encontrar.

Os dados fornecidos de classificação de dados tratam os candidatos como classificação binária, visto que os custos de classificação com múltiplas classes seriam de um elevado custo, nesta área de trabalho. Neste *dataset*, pulsares legítimos são uma minoria, visto que, como podemos ver, temos apenas 1639 verdadeiros pulsares e 16259 casos de falsos positivos causados por *RFI/noise*, no total de 17898 candidatos.

2.4 Abordagem

2.4.1 Modelo de aprendizagem a aplicar

Multilayer Perception (MLP) é uma rede neuronal artificial do tipo *feedforward*, isto é, as conexões entre os neurónios não formam ciclos o que faz com que a informação se mova num único sentido. Esta rede consiste em pelo menos três camadas de nós, sendo então uma rede neuronal multicamada. Além disso, exceto os nós de entrada, cada nó é um neurónio que utiliza a função de ativação não linear. O MLP aplica uma técnica de aprendizagem supervisionada indutiva denominada *backpropagation*.

A aprendizagem supervisionada é uma tarefa de *machine learning* que deduz uma função a partir de um conjunto de dados de treino categorizados. Este tipo de aprendizagem indutiva demonstra ser o mais apropriado para o trabalho, visto que analisa os dados de entrada, os candidatos, e obtém uma função que permite identificar um pulsar, através de aproximações.

Como mencionado anteriormente, ***backpropagation*** *(backward propagation of errors)* é um algoritmo utilizado por esta aprendizagem, onde a rede opera em duas fases e implementa um método de otimização chamado *gradient descent*. Primeiramente, um padrão é apresentado à camada de entrada, e propaga-se, camada a camada, até à última camada da rede. De seguida, o resultado obtido é comparado ao desejado. Se estiver errado, um valor de erro é calculado para cada um dos neurónios da camada de saída. Este valor é propagado por retrocesso, até à camada de entrada, e os pesos das arestas da rede neuronal são atualizados de forma a minimizar a função de perda.

Assim, após serem recebidos dados de entrada suficientes, a rede converge para um estado onde os erros são cada vez menores, ou seja, aprendeu.

2.4.2 Arquitetura das Redes Neuronais

Na realização deste trabalho vai ser usada uma **rede *feedforward* de múltiplas camadas**. Isto implica que a rede tenha vários nós organizados nas seguintes camadas:

1. Uma camada chamada **Input Layer**, que recebe os dados do exterior e os introduz à rede. Não é necessária nenhuma computação nesta camada, pois os nós só têm a responsabilidade de transferir informação para as camadas seguintes.
2. Uma ou mais camadas chamadas **Hidden Layers**, que não têm qualquer ligação ao exterior da rede. Elas realizam a computação da informação proveniente da Input Layer e passam-na para a Ouput Layer.
3. Uma camada chamada **Output Layer**, que é responsável pela computação dos dados provenientes das Hidden Layers e transferência destes para o exterior da rede.

Numa rede *feedforward* a informação move-se unicamente num sentido (da Input Layer para as Hidden Layers e de seguida para a Output Layer), ou seja, cada nó só pode estar ligado a nós da camada seguinte e estas ligações entre nós não fazem ciclos.

3 Desenvolvimento

3.1 Ferramentas utilizadas

Na implementação deste projeto recorremos às seguintes ferramentas:

- **Python**: linguagem de programação com um grande número de bibliotecas disponíveis para o tema em questão.
- **Keras**: uma biblioteca de redes neuronais *open-source*, escrita em *Python*.
- **Tensorflow with GPU support**: uma biblioteca *open-source* para programação de fluxo de dados em várias tarefas. É também utilizado para aplicações de *machine learning*, como as redes neuronais aqui apresentadas. Optámos por utilizar a GPU para que o programa demore o menor tempo possível a executar.
- **Scikit-learn**: uma biblioteca de *machine learning* com acesso a inúmeros algoritmos úteis.
- **CUDA Toolkit 9.0**: um ambiente de desenvolvimento para criar aplicação de alta performance aceleradas pela GPU.
- **cuDNN 7.0**: NVIDIA CUDA® Deep Neural Network library (cuDNN) é uma biblioteca acelerada pela GPU de primitivas para redes neuronais.

3.2 Pré-Processamento do Dataset

Cada candidato é descrito por 8 variáveis continuas, bem como uma variável que indica se esse candidato é ou não um verdadeiro pulsar. As primeiro quatro varáveis foram obtidas pelo perfil integrado, isto é, são variáveis que tiveram em consideração o tempo e a frequência para o seu cálculo. Enquanto que as seguintes quatro foram obtidas de uma forma similar, no entanto, através de uma curva DM-SNR, proveniente de uma passagem para as duas dimensões de uma matriz representada pela *Dispersion Measure*, pelo *Signal to Noise Ratio* e pelo *offset* do período, tal como podemos observar no paper original de Robert James Lyon, capítulo 2.2.1, tal como podemos ver na imagem imediatamente abaixo.

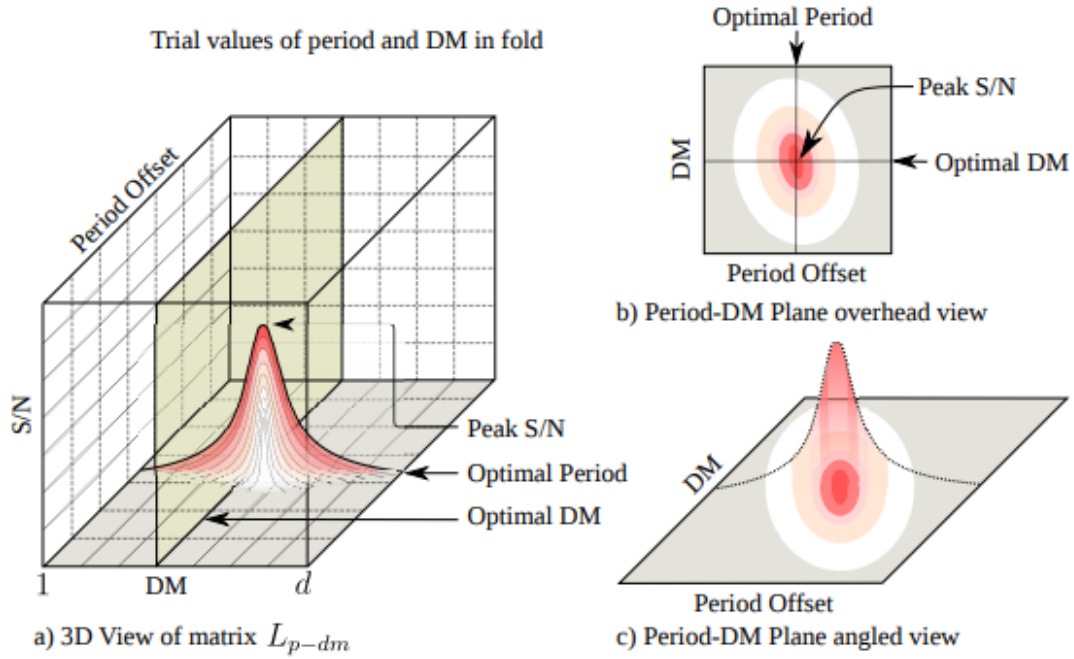

Figura 1: Visualização dos dados recolhidos para cada candidato a pulsar. Em a) podemos observar um pico que representa o maior *Signal to Noise Ratio*. Em b) temos uma visão da matriz de cima, que nos ajuda a distinguir pulsares, visto que estes devem ter regiões circulares de SNR crescente, à medida que os valores do período e do *Dispersion Measure* ficam ótimos. Já em c) temos uma visão lateral de b)

Podemos então observar uma lista ordenada das várias características de cada candidato, as quais estarão a ser tomadas em consideração no processamento dos dados.

1. Mean of the integrated profile.
2. Standard deviation of the integrated profile.
3. Excess kurtosis of the integrated profile.
4. Skewness of the integrated profile.
5. Mean of the DM-SNR curve.
6. Standard deviation of the DM-SNR curve.
7. Excess kurtosis of the DM-SNR curve.
8. Skewness of the DM-SNR curve.

Face a esta notória ausência de dados no dataset providenciado, teríamos, teoricamente, de proceder a algumas táticas para melhorar o dataset para, igualmente, tornarmos o resultado o mais preciso possível. Procedemos então à descoberta e implementação de como abordarmos este problema. Mudar a métrica de medição de performance, seria algo que não se aplica neste caso, pois o objetivo é, de facto, que seja o mais precisa possível a detetar se é ou não pulsar; procedemos então à análise do *resampling* dos dados, ora com cópias de pulsares verdadeiros, ora como eliminação de falsos positivos, no entanto, isto não seria ideal por si só, visto que o número de dados é bastante elevado e alterações manuais nos mesmos seria muito trabalhosa.

Procedemos então à alternativa que nos pareceu mais viável que será gerar candidatos sintéticos, mas não aleatórios, utilizando, por isso, alguns algoritmos, da biblioteca scikit-learn, biblioteca de *machine learning* que incluí algoritmos de classificação, regressão, e de *clustering*. Por exemplo, testámos alguns algoritmos de over sampling, como o SMOTE (Synthetic Minority Over-sampling Technique), no entanto, como o dataset já é bastante grande, testámos, também, alguns algoritmos de *under sampling*, como o *Random Under Sampler*, o *Neighbourhood Cleaning Rule* , os algoritmos *ENN, RENN e All-KNN*, bem como o algoritmo *Cluster centroids*. No capítulo 4 iremos avaliar a performance de cada um destes algoritmos mais à frente, bem como o utilizado para o resultado final.

3.3 Estrutura da aplicação

De uma forma geral, a nossa aplicação pode ser dividida em três partes principais.

- Carregamento de dados;
- Configuração da rede;
- Treino e teste da rede.

3.3.1 Carregamento de dados

Após os dados serem carregados, é chamada a função split_train_dataset(), que baralha aleatoriamente os conjuntos de dados, por forma a que a sua introdução na rede seja aleatória e equilibrada. Aproveitando essa mesma função para dividir o *dataset* nas duas partes necessárias, os dados de teste e os dados de treino, com uma percentagem de dados de teste a indicar pelo argumento *test_ratio*, sendo que estes são aleatórios, visto que anteriormente já baralhámos os vários dados.

```
1 # split training dataset
2 def split_train_dataset(data, test_ratio):
```

```
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(test_ratio * len(data))
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data[train_indices,:], data[test_indices,:]

dataframe = np.loadtxt('../dataset/HTRU_2.csv', delimiter=',', dtype=np.
    float64)

train, test = split_train_dataset(dataframe, 0.2)

# generate training data
x_train, y_train = train[:,:8], train[:,8]

# generate test data
x_test, y_test = test[:,:8], test[:,8]
```

Listing 1: Carregamento do dataset.

Posteriormente, passamos a escolher o método de sampling pretendido através da função $sampling(algorithm, x\_train, y\_train)$, onde algorithm é uma String para podermos escolher entre os vários algoritmos implementados, podendo este ser:

1. Standard Scaler;
2. Random Under Sampling;
3. SMOTE regular;
4. SMOTE borderline1;
5. SMOTE borderline2;
6. SMOTE svm;
7. Neighbourhood Cleaning Rule;
8. EditedNearestNeighbours;
9. RepeatedEditedNearestNeighbours;
10. All-K-Nearest-Neighbors;
11. Cluster Centroids;
12. Cluster Centroids with Hard Voting;
13. No Sampling.

Apesar de tudo, o código está modular, de modo que qualquer outro algoritmo poderá ser, posteriormente, adicionado sem qualquer problema, no entanto, o algoritmo que

atingiu melhor *precision* foi a ausência do mesmo, de maneira que, a rede neuronal final, não incluí nenhum dos algoritmos.

3.3.2 Configuração da rede

Apesar de ser difícil, antes de fazer qualquer teste, prever qual a melhor configuração da rede, a nossa conclusão no relatório intermédio foi bastante próxima da que viríamos a implementar na solução final, ie, a seguinte configuração.

```
# model creation and configuration
model = Sequential()

model.add(Dense(9, input_dim=data_dim, kernel_initializer='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, kernel_initializer='uniform'))
model.add(LeakyReLU(alpha=0.3))
model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=["accuracy"])
```

Listing 2: Configuração da rede.

3.3.3 Input layer

O número de nós nesta camada é geralmente o número de atributos do *dataset* usado, no caso deste problema 8. Decidiu-se a estes nós adicionar um nó *bias* para ajudar a rede a ajustar-se aos dados recebidos, a camada fica então com **9 neurónios**.

Um **nó *bias*** é um neurónio extra para cada camada da rede exceto a output layer e que guarda o valor de 1. Este nó não está ligado a nenhuma camada anterior à que se encontra e, portanto, não representa uma verdadeira atividade. Sem este nó, num input com todas as variáveis a zero o output seria tudo zero também, esta pode ser uma boa solução para alguns sistemas, mas para a maioria é demasiado restrita. Na fase de ajustes da rede, muda-se simultaneamente o peso e o valor, portanto uma mudança no peso pode neutralizar a mudança no valor que foi útil para uma instância de dados prévia. A adição de um nó *bias* ajuda no controlo do comportamento de cada camada, pois permite mover a função de transferência horizontalmente ao longo do eixo de input, continuando com a forma/curvatura inalterada. Assim, conseguem-se diferentes resultados e pode-se alterar conforme as necessidades do problema.

Para esta camada iremos, também, aplicar uma função de ativação tangencial hiperbólica, pois, desta maneira temos uma função diferencial que promove maiores gradientes. Aplicamos ainda uma camada de *dropout* para diminuir a percentagem de erro de classificação.

3.3.4 Hidden layers

O primeiro problema para determinar a configuração desta parte da rede é determinar o número de camadas que a constituem e para isso é importante ter em conta a performance da rede. As situações em que a performance aumenta com a adição de uma segunda ou terceira camada são raras, pois na maioria dos problemas uma Hidden Layer é suficiente, portanto a nossa rede terá **uma só camada deste tipo**.

O segundo problema é determinar quantos nós a Hidden Layer terá. Para tal fomos testando os vários valores até encontrarmos um que satisfizesse os nosso requerimentos ao nível da *accuracy*, desta maneira, verificámos que 32 nós serão suficientes e darão boas capacidades de seleção ao modelo, mesmo em amostras que não estejam presentes neste conjunto.

O terceiro e último problema é determinar que função é usada para assumir o papel de função de transferência. Esta função é usada para determinar o output de cada neurónio. Foram consideradas dois tipos de funções: Sigmoid e ReLU.

A função Sigmoid tem um domínio entre 0 e 1 e tem como expressão:

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Enquanto que a função ReLU tem um domínio de 0 a infinito e expressão:

$$R(x) = x^+ = max(0, x)$$

O problema da função Sigmoid é que o máximo da sua derivada é igual a 1/4, ou seja, quando se aumenta o valor de x, o gradiente diminui exponencialmente. Como o gradiente da função ReLU é 1 ou 0, esta não tem este problema.

Foi então decidido usar esta última função **ReLU como função de transferência**. No entanto, com a investigação e desenvolvimento, descobrimos uma função de ativação que permitiria um pequeno gradiente diferente de zero, caso a unidade esteja saturada e não ativa, o **LeakyReLU**. Tal como o paper Rectifier Nonlinearities Improve Neural Network Acoustic Models o pôde comprovar, nós também conseguimos atingir uma maior performance com esta função, visto que, esta converge mais rapidamente, apresentando, no final, claros resultados superiores.

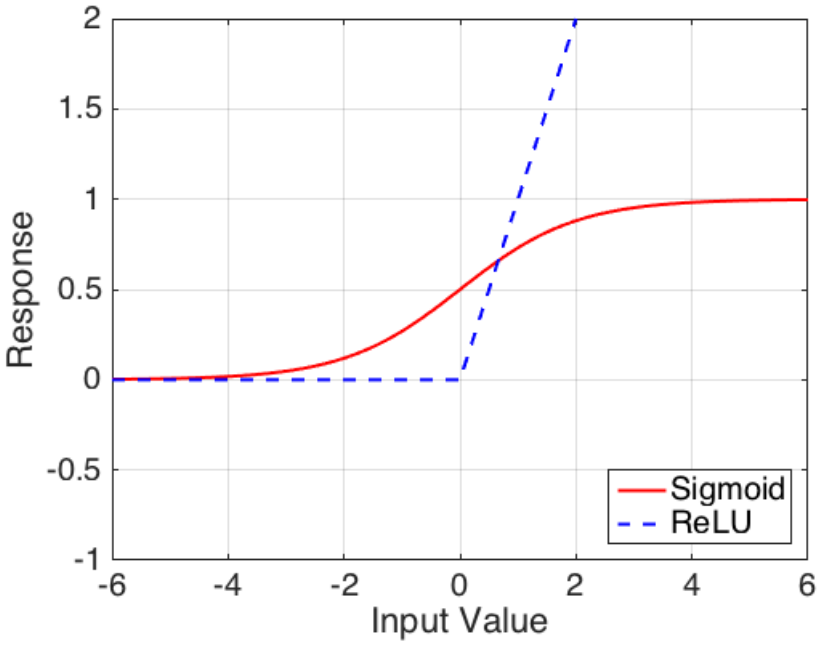

Figura 2: Funções Sigmoid e ReLU

3.3.5 Output layer

Para determinar o número de neurónios nesta camada considerámos duas opções:

1. Usar um único nó de output e a função de transferência deste seria uma função Sigmoid. Dependendo do resultado desta função ser menor ou maior que 0.5 concluiríamos se os dados fornecidos no input correspondiam ou não a um Pulsar.
2. Usar dois nós que representariam as classes Pulsar e Não-Pulsar. O resultado da função de transferência de cada um diria a sua probabilidade de se verificar sendo que a soma dos dois daria 1 (ex.: Pulsar:0.70 e Não-Pulsar:0.30 concluindo assim que os dados correspondiam a um Pulsar). Esta função de transferência seria do tipo Softmax. Esta função achata a unidade de cada output entre 0 a 1 tal como uma função Sigmoid, mas também divide cada output de modo a que a soma deles dê 1. O output desta função diz-nos a probabilidade de cada uma das classes. A expressão da função da Softmax é:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad for \quad j = 1, ..., K$$

Foram tidos estes dois tipos de funções em consideração pois são as duas usadas em casos onde se pretende prever a probabilidade como output. Se como output fosse preciso mais do que uma classe, seria obrigatório usar Softmax, mas como o resultado é binário (é ou não é pulsar) **vai ser usada a função Sigmoid** pois matematicamente é mais simples.

3.3.6 Treino e teste da rede

Para o treino e teste da rede utilizamos os dados que dividimos anteriormente utilizando a função *fit()* da biblioteca Keras, ao longo de 5 *epochs* ao qual chegámos através de testes, a um *batch size* de 128. Para além disso, aproveitámos das ferramentas que tivémos acesso e utilizámos o TensorBoard, para termos acesso a algumas estatísticas sobre os vários testes realizados.

```
tensorboard = TensorBoard(log_dir='./../Graph', histogram_freq=0,
    write_graph=True, write_images=True)

# model fitting
model.fit(x_train, y_train,
          epochs=5,
          batch_size=batch_size,
          callbacks=[tensorboard])

# precision calculation
pred = model.predict(x_test, batch_size=batch_size)

y_pred = [i[0] for i in pred]

def precision_calculation(a):
    if a > 0.5:
        return 1.0
    return 0.0

vfunc = np.vectorize(precision_calculation)
y = vfunc(y_pred)

# reporting and printing summary, scores and some stats
report = classification_report(y_test, y)

print(report)

score = model.evaluate(x_test, y_test, batch_size=batch_size)
print(model.summary())
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Listing 3: Treino e teste da rede.

Já para obtermos os resultados da rede neuronal, utilizámos a função $predict(x\_test, batch\_size = batch\_size)$, que retorna uma previsão de resultados possíveis entre 0.0 e 1.0. Com estes valores igualamos a 1 qualquer valor superior a 0.5 e a 0 qualquer valor igual ou inferior a 0.5, e, com a lista resultante, comparamos com os valores reais através da função $classification\_report(y\_test, y)$, que retorna algo como o seguinte:

```
             precision    recall  f1-score   support

        0.0       0.98      1.00      0.99      3259
        1.0       0.94      0.78      0.85       320

avg / total       0.98      0.98      0.97      3579
```

Listing 4: Exemplo de output da precisão por classe.

Extraímos ainda a *accuracy* geral através da função $evaluate(x\_test, y\_test, batch\_size = batch\_size)$, onde obtemos um resultado similar ao seguinte:

```
Test loss: 0.085446722183057
Test accuracy: 0.9759709411541421
```

Listing 5: Exemplo de output da precisão geral.

3.4 Detalhes relevantes da implementação

Relativamente à implementação das redes neuronais e funcionalidades a estas associadas, optamos por recorrer às várias bibliotecas para *Python*, que se demonstraram ser bastante eficientes, apesar de estarem claramente sub-desenvolvidas no que toca a documentação. Apesar de tudo, a nossa implementação acabou por ser bastante precisa, mas também, flexível, de maneira que é bastante fácil a escolha de qualquer um dos algoritmos de *sampling* com a possibilidade de visualização do resultado de alguns.

A utilização de uma função de divisão do *dataset* também foi bastante benéfica, visto que posteriormente nos poupou trabalho, para além de assegurar a imparcialidade dos dados. Para além disso, a impressão de dados importantes da rede neuronal, como a comparação dos dados reais com a previsão, o sumário do modelo e o resultado final da rede neuronal, revelaram-se bastante importantes durante a fase de desenvolvimento, de maneira que tivemos um melhor acesso ao estado atual da rede neuronal, para desenvolvermos sobre cada iteração da mesma. Para concluir, o registo de cada modelo após cada execução, ajuda a ter um histórico das várias precisões obtidas ao longo do desenvolvimento.

4 Experiências

4.1 Experiência 1: Número de *Epochs*

A determinação do número de *Epochs* foi relativamente direta, visto que por mais *epochs* que utilizássemos, a precisão estabilizava a partir da terceira *epoch*, tal como podemos ver na seguinte figura, de maneira que optámos por utilizar 5, para simplesmente termos uma maior certeza que não iria variar.

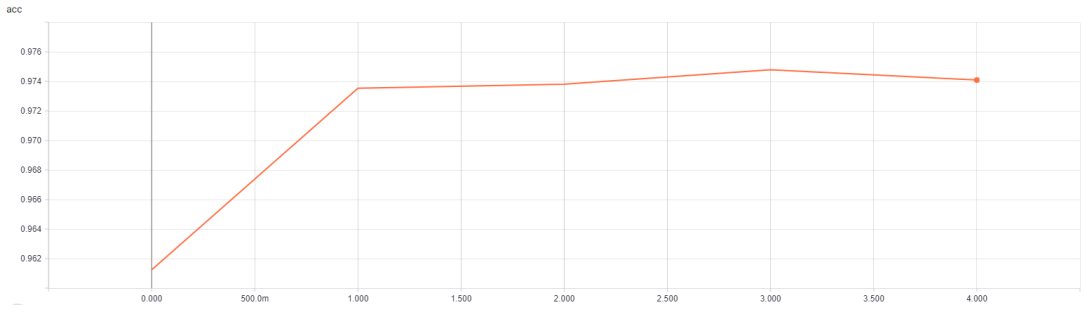

Figura 3: Accuracy over Time

4.2 Experiência 2: ReLU VS LeakyReLU

Através da leitura do paper orignário no departamento de Ciência de Computadores na Universidade de Stanford, e, como é óbvio através dos testes demonstrados abaixo, podemos observar que a função de ativação LeakyReLU acabar por ter uma precisão superior à ReLU base, de maneira que acabámos por implementar a mesma no configuração final.

```
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.99      3259
         1.0       0.93      0.78      0.85       320

avg / total       0.97      0.98      0.97      3579

Test loss: 0.08589417685357244
Test accuracy: 0.9754121258426026
```

Listing 6: Precisão da função de ativação ReLU.

```
              precision    recall  f1-score   support

         0.0       0.98      1.00      0.99      3259
```

```
4           1.0       0.94      0.78      0.85       320
5
6  avg / total       0.98      0.98      0.97      3579
7
8  Test loss: 0.08544672622581406
9  Test accuracy: 0.9759709411541421
```

Listing 7: Precisão da função de ativação LeakyReLU.

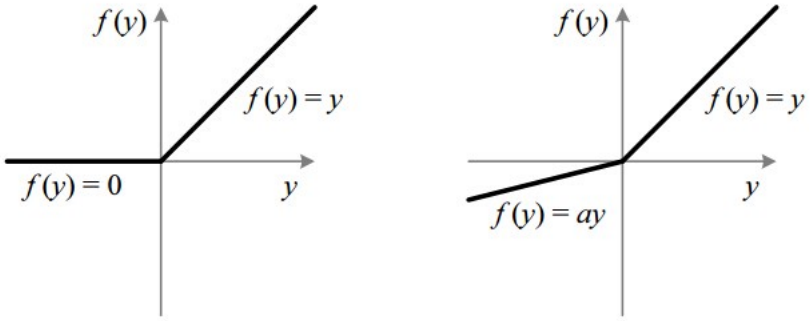

Figura 4: Leaky ReLU VS ReLU.

Tal como foi falado anteriormente, obtivemos portanto uma ligeira melhoria na precisão pelo facto de esta função de ativação convergir mais rapidamente, devido ao facto de as Leaky ReLUs serem uma tentativa de reparar o problemas das *"dying ReLU's"* Já que em vez de a função igualar a zero quando x ¡ 0, esta terá um pequeno declive negativo de, por exemplo, 0.01. Isto é, a função passa a ser $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$, onde α é uma pequena constante. Apesar de tudo, esta função não é intrinsecamente melhor que a ReLU, no entanto, para o este caso, demonstrou sê-lo.

4.3 Experiência 3: *Optimizer*

Para esta experiência, com o conhecimento prévio através da pesquisa realizada ao longo do desenvolvimento sabíamos que, para este caso de classificação binária, iríamos utilizar ora o *optimizer* Adam, ora iríamos utilizar o *optimizer* RMSprop, no entanto, por desejo de experimentar e descobrir acabámos por testar todos os *optimizers*, onde obtivémos os seguintes resultados:

1. Overall Test accuracy: 0.9709416036167502 with **SGD** optimizer.

   Positive Class precision: 0.94

2. Overall Test accuracy: 0.9759709411541421 with **RMSprop** optimizer.

   Positive Class precision: 0.94

3. Overall Test accuracy: 0.9754121258426026 with **Adagrad** optimizer.

   Positive Class precision: 0.92

4. Overall Test accuracy: 0.9748533105310633 with **Adadelta** optimizer.

   Positive Class precision: 0.93

5. Overall Test accuracy: 0.9765297564656816 with **Adam** optimizer.

   Positive Class precision: 0.92

6. Overall Test accuracy: 0.9751327181868328 with **Adamax** optimizer.

   Positive Class precision: 0.90

7. Overall Test accuracy: 0.9748533105310632 with **Nadam** optimizer.

   Positive Class precision: 0.92

Desta maneira, apesar de ser claro que o **Adam** possui uma maior precisão geral, no que toca a detetar se é ou não um pulsar o optimizador **SGD** ou **RMSprop** são claros vencedores, porém, o **RMSprop** possui maior precisão geral, de maneira que optámos por utilizar este na rede neuronal.

4.4 Experiência 4: *Sampling*

Para a nossa quarta e última experiência, testámos os vários algoritmos que implementámos no nosso projeto como podemos ver no seguinte excerto:

```
Test accuracy: 0.9759709411541421 with no sampling.
             precision    recall  f1-score   support

        0.0       0.98      1.00      0.99      3259
        1.0       0.94      0.78      0.85       320

avg / total       0.98      0.98      0.97      3579

_______

Test accuracy: 0.9463537301754745 with Standard Scaler.
             precision    recall  f1-score   support

        0.0       0.98      0.96      0.97      3259
        1.0       0.66      0.84      0.74       320
```

```
16
17 avg / total       0.95      0.95      0.95      3579
18
19 ———————
20
21 Test accuracy: 0.9491478067331722 with Random Under Sampling.
22              precision    recall  f1-score   support
23
24         0.0       0.99      0.95      0.97      3259
25         1.0       0.66      0.90      0.76       320
26
27 avg / total       0.96      0.95      0.95      3579
28
29 ———————
30
31 Test accuracy: 0.917015925869991 with SMOTE Regular.
32              precision    recall  f1-score   support
33
34         0.0       0.99      0.92      0.95      3259
35         1.0       0.52      0.93      0.67       320
36
37 avg / total       0.95      0.92      0.93      3579
38
39 ———————
40
41 Test accuracy: 0.9145012569680631 with SMOTE Borderline1
42              precision    recall  f1-score   support
43
44         0.0       0.99      0.91      0.95      3259
45         1.0       0.51      0.93      0.66       320
46
47 avg / total       0.95      0.91      0.93      3579
48
49 ———————
50
51 Test accuracy: 0.9022073206471211 with SMOTE Borderline2
52              precision    recall  f1-score   support
53
54         0.0       0.99      0.90      0.94      3259
55         1.0       0.48      0.94      0.63       320
56
57 avg / total       0.95      0.90      0.92      3579
58
59 ———————
60
61 Test accuracy: 0.9304274943295255 with SMOTE svm
62              precision    recall  f1-score   support
63
64         0.0       0.99      0.93      0.96      3259
```

```
        1.0       0.57      0.93      0.70       320

avg / total       0.95      0.93      0.94      3579

-------

Test accuracy: 0.9734562722522142 with Neighbourhood Cleaning Rule.
             precision    recall  f1-score   support

        0.0       0.99      0.98      0.99      3259
        1.0       0.84      0.87      0.85       320

avg / total       0.97      0.97      0.97      3579

-------

Test accuracy: 0.9754121258426026 with ENN.
             precision    recall  f1-score   support

        0.0       0.98      0.99      0.99      3259
        1.0       0.88      0.84      0.86       320

avg / total       0.97      0.98      0.98      3579

-------

Test accuracy: 0.9740150875637538 with RENN.
             precision    recall  f1-score   support

        0.0       0.99      0.98      0.99      3259
        1.0       0.85      0.87      0.86       320

avg / total       0.97      0.97      0.97      3579

-------

Test accuracy: 0.9759709411541421 with AllKNN.
             precision    recall  f1-score   support

        0.0       0.98      0.99      0.99      3259
        1.0       0.92      0.80      0.86       320

avg / total       0.98      0.98      0.98      3579

-------

Test accuracy: 0.9455155072081652 with Cluster Centroids.
             precision    recall  f1-score   support

```

```
114          0.0       0.99      0.95      0.97      3259
115          1.0       0.64      0.90      0.75       320
116
117  avg / total       0.96      0.95      0.95      3579
118
119  ----------
120
121  Test accuracy: 0.9670298961695095 with Cluster Centroids with Hard Voting.
122                precision    recall  f1-score   support
123
124          0.0       0.99      0.98      0.98      3259
125          1.0       0.79      0.86      0.82       320
126
127  avg / total       0.97      0.97      0.97      3579
```

Listing 8: Sampling.

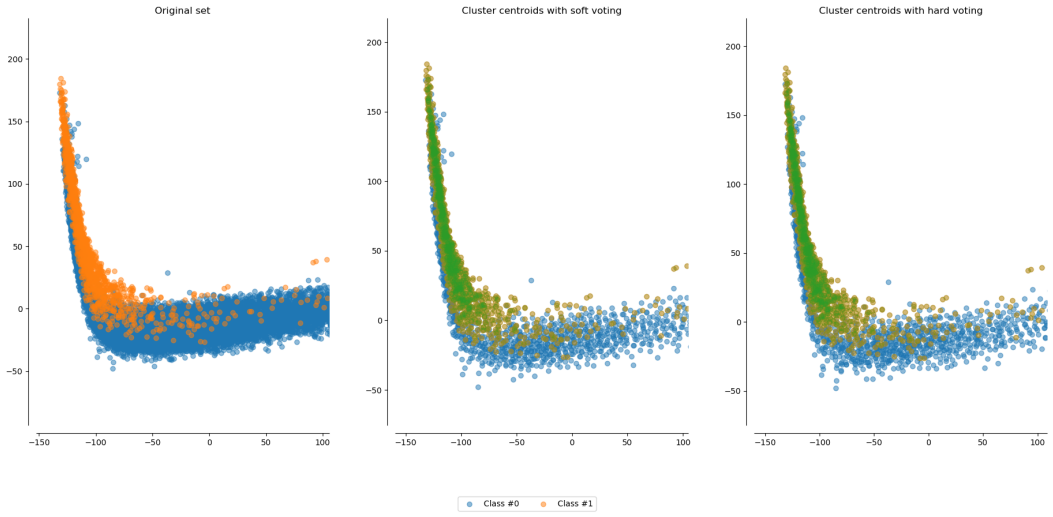

Figura 5: Clustering.

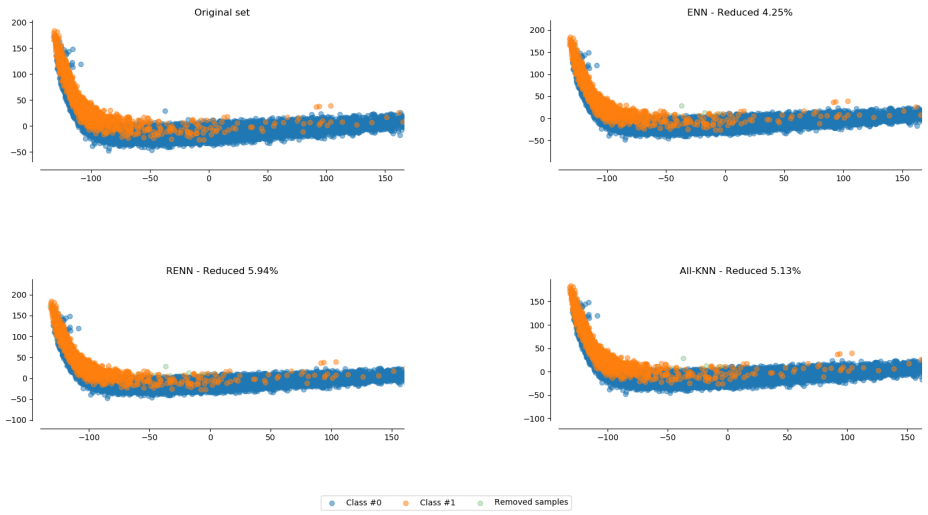

Figura 6: ENN, RENN, AllKNN.

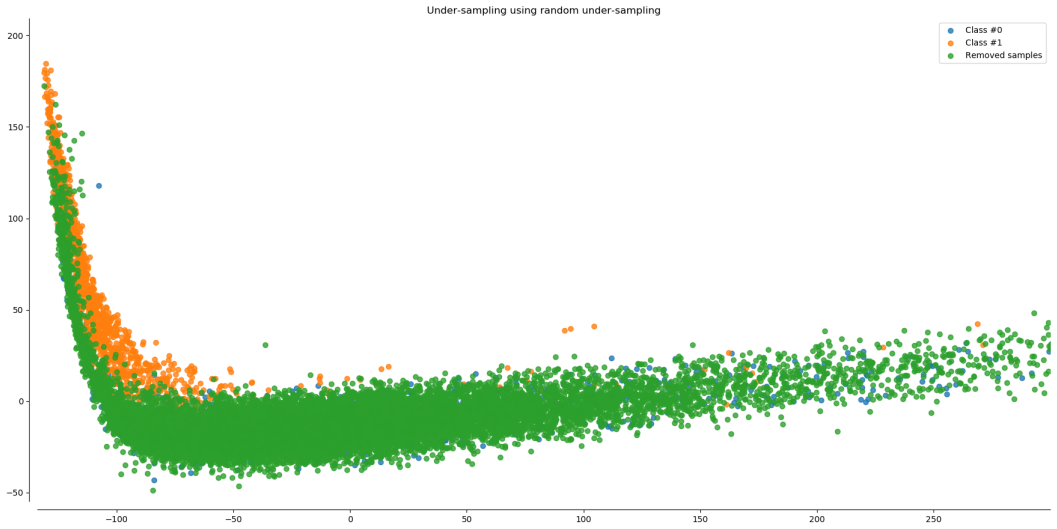

Figura 7: Random Under Sampling.

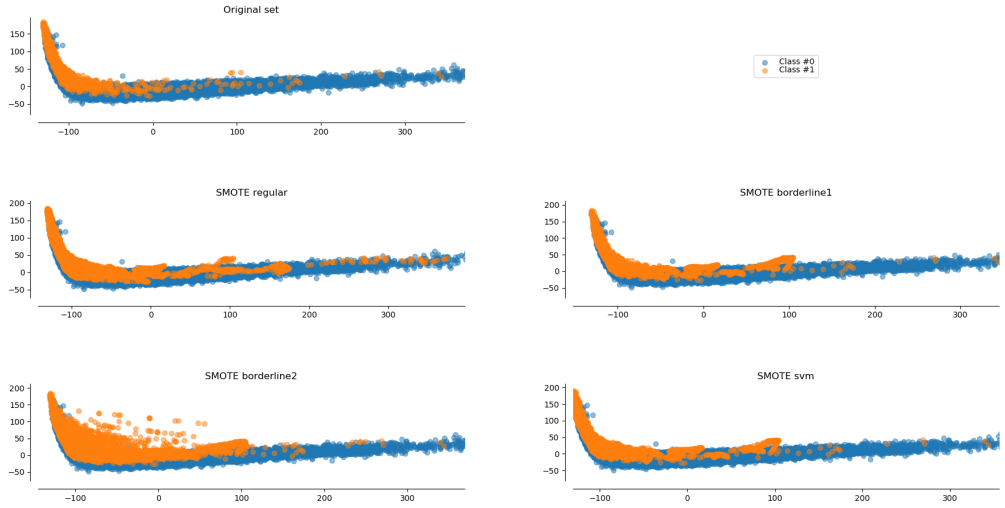

Figura 8: SMOTE.

Analisando estes resultados, podemos verificar que muitos dos algoritmos, como seria esperado, conseguem atingir uma boa precisão geral, no entanto, o caso que apresenta uma melhor precisão para o caso de ser um pulsar é, claramente, quando não fazemos *sampling* aos dados. No entanto, o algoritmo All-KNN (All-K-Nearest-Neighbors) apresenta também uma boa precisão, pelo que seria, sem dúvida a segunda escolha.

5 Conclusões

O facto de o tema deste trabalho reunir grandes matérias da inteligência artificial, nomeadamente redes neuronais, com a astronomia torna a sua implementação e compreensão mais interessante. A aplicação da matéria aprendida nas aulas, tanto teóricas como práticas, contribuiu para um melhor aproveitamento e aprofundamento do tema em questão.

No seguimento do capitulo anterior, podemos concluir que a realização de experiências nos permitiu perceber melhor o funcionamento da aplicação e que fatores influenciam os seus resultados. Também, é de realçar que, durante toda a implementação, notamos que o aumento do número de neurónios, para além dos recomendados, e o aumento do número de camadas intermédias, não influenciavam significativamente os resultados obtidos.

Em suma, o trabalho foi finalizado com sucesso, tendo-se cumprido todos os objetivos e ultrapassado todos os obstáculos.

6 Melhoramentos

O desempenho da rede neuronal apresenta-se bastante satisfatório, com uma taxa de precisão sempre acima dos 97%. No entanto, um melhoramento possível seria ao nível do *dataset* providenciado, visto se ter notado uma ausência de dados relevantes, que nos obrigou a proceder a algumas táticas para o melhorar.

7 Recursos

7.1 Bibliografia

1. Scikit-Learn
2. Machine Learning Book
3. Andrew L. Maas,Awni Y. Hannun, Andrew Y. Ng - Rectifier Nonlinearities Improve Neural Network Acoustic Models
4. Ryan S. Lynch, Searching for and Identifying Pulsars
5. Keras Documentation
6. Imbalanced Learn - Scikit-Learn
7. Robert James Lyon, Why are Pulsars Hard to Find?
8. CS231n Convolutional Neural Networks for Visual Recognition
9. 8 Tactics to Combat Imbalanced Classes in Your Machine Learning Dataset (APAGAR?)
10. ReLU and Softmax Activation Functions
11. How to choose the number of hidden layers and nodes in a feedforward neural network?
12. Role of Bias in Neural Networks
13. What is bias in artificial neural network?
14. Why are bias nodes used in neural networks?
15. A Quick Introduction to Neural Networks
16. Sigmoid vs Relu function in Convnets
17. What is special about rectifier neural units used in NN learning?
18. Perceptron Multi-Camadas (MLP)
19. Apontamentos das aulas teóricas de Inteligência Artificial

7.2 Software

Software utilizado no projeto:

- Visual Studio Code
- Linguagem de programação Python
- Bibliotecas: Tensorflow, Keras, Scikit-learn

7.3 Elementos do grupo

Percentagem aproximada de trabalho efetivo de cada elemento do grupo:

- Afonso Jorge Ramos: 33%
- Bárbara Sofia Silva: 33%
- Julieta Pintado Jorge Frade: 33%

8 Apêndice

8.1 Manual de utilizador

1. Assegurar que todas as dependências do Tensorflow estão instaladas.
2. Proceder à instalação das Bibliotecas utilizadas.

```
1    pip install keras
2    pip install -U imbalanced-learn
3    pip install -U scikit-learn
```

3. Proceder à execução do código.

   py .\ pulsars.py **<argumento opcional contento sampling pretendido><argumento opcional contento modo de SMOTE pretendido>**