



Computação Natural
**Redes Neurais Artificiais em Exames
Clínicos**

Hugo Gonçalves (PG38928)
Luis Bouça (PG38933)
Diogo Rodrigues (PG37150)
Henrique Ribeiro (PG38415)

Março, 2019

Área de Aplicação: Deep Neural Network
Palavras-chave: *GA, ANN, Tensorflow, K-fold*

Conteúdo

Introdução	1
1 <i>Conjunto de Dados</i>	2
1.1 Tratamento de Dados	2
2 Redes Neurais	3
2.1 Modelo	4
3 Validação	5
3.1 Particionamento Treino/Teste	5
3.2 Validação Cruzada K-Fold	5
4 Algoritmos Genéticos	6
4.1 População	7
4.2 Aptidão	8
4.3 Pais	9
4.4 Cruzamento	10
4.5 Mutação	11
4.6 Paragem	12
5 Resultados	12
6 Conclusão	13

Lista de Figuras

1	Normalização	3
2	Estimadores	3
3	Classificador	4
4	Treino da rede	4
5	Teste da rede	4
6	TrainTestSplit	5
7	Validação Cruzada	5
8	Parâmetros do algoritmo	6
9	População inicial	7
10	Função de aptidão	8
11	Escolha dos pais	9
12	Função de Cruzamento	10
13	Função de Mutação	11
14	Resultado com 10 gerações	12
15	Resultado com 2 gerações	12

Introdução

Redes neuronais podem ser aplicadas em diversas áreas, como economia saúde, segurança, entre outros. O objetivo deste trabalho prático passa pela criação de um modelo capaz de indicar o tipo de cancro num exame de mamografia, usando para isso redes neuronais artificiais.

A importância que estes modelos trazem possibilitam uma maior rapidez na detecção do cancro, permitindo assim uma ação mais atempada e rápida para com o paciente.

A construção do modelo implicou algumas técnicas de otimização do mesmo, a fim de o tornar mais robusto, tanto na sua fase de treino como na sua fase de teste. Foram assim implementadas técnicas de validação cruzada, e algoritmos genéticos, que permitiram preparar o modelo para novos casos e criar uma rede que esteja adaptada ao problema que estamos a tentar solucionar.

O objetivo com esta rede, é que o desempenho da mesma seja suficientemente bom para que os seus resultados sejam confiáveis, isto porque o erro neste tipo de situações é bastante importante, ou seja, casos de **True Negatives** e **False Positives** podem significar a vida de um doente.

1 *Conjunto de Dados*

O conjunto de dados fornecido para a realização deste trabalho prático é o **mammographic_masses**. Este dataset é assim composto por 961 instâncias, contendo 6 atributos. Os atributos são definidos por:

- **BI-RADS:** Acrônimo para Breast Imaging-Reporting and Data System. Contem valores que variam entre 1 e 5, e funcionam como classificação num exame de mamografia, ou seja, a possibilidade de existência de cancro num exame.
- **Age:** Idade do paciente, constituído por um valor inteiro.
- **Shape:** Refere o formato do tipo de cancro, sendo constituída por valores que variam entre 1 e 4 (Redondo, ovular, lobular, irregular).
- **Margin:** Referente à massa da margem, contendo valores que variam entre 1 e 5 (circunscrito, microlobulado, obscurecido, mal definido, espiculado).
- **Density:** Valor referente densidade da massa constituída pelo cancro, contendo valores que variam entre 1 e 4 (alta, média, baixa, contendo gordura).
- **Severity:** Classificação quanto ao tipo de cancro, constituída pelos valores 0 e 1 (benigno, maligno).

Quando exposta ao conjunto de dados fornecido, depois de treinada, a machine learning deve ter a capacidade para classificar os valores 0 ou 1 referentes ao tributo **Severity**.

1.1 Tratamento de Dados

Antes de proceder ao treino da machine learning, foi feito um processo de tratamento aos dados fornecidos pelo dataset. O objeto é validar os mesmos, removendo impurezas nos dados, e preparando a machine learning para que o processo de treino seja mais eficaz.

O processo de tratamento de dados é constituído inicialmente pela remoção de instâncias que contenham valores em falta em qualquer um dos seus atributos.

Posteriormente procedesse à remoção do atributo **BI-RADS**, isto porque se pode concluir que o mesmo não está diretamente associado à ocorrência de qualquer umas das possíveis classes do atributo **Severity**. Uma das possibilidades seria confirmar recorrendo a um feature selection, quais dos atributos têm um maior peso na representação de um componente PCA.

Finalmente, devido à existência de dados com diferentes escalas, foi feita uma normalização aos mesmos.

```
#dataset normalization
def Normalization(dataset):
    scaler = StandardScaler()
    scaler = scaler.fit_transform(dataset.drop('Severity',axis=1))
    dataset = pd.DataFrame(scaler,columns=dataset.columns[:-1])
    return dataset
```

Figura 1: Normalização

O excerto de código presente na figura 1, mostra o processo de normalização, no qual a classe **Severity** foi excluída deste processo, devido ao facto de que a mesma é constituída por 2 classes possíveis de serem classificadas.

2 Redes Neurais

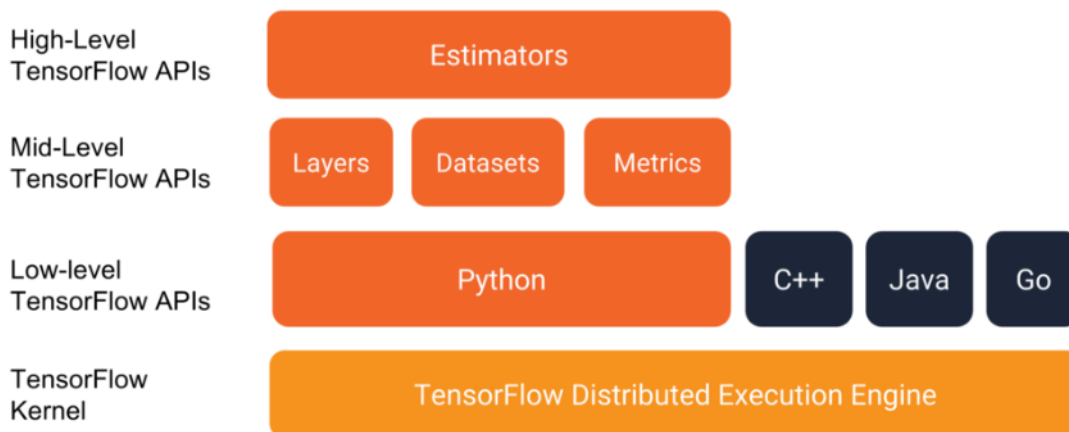


Figura 2: Estimadores

A construção da rede teve por base o uso de estimadores, que são APIs de alto nível do tensorflow, conforme é possível observar na figura 2.

A vantagem no uso deste tipo de APIs, é que as mesmas apresentam todas as funcionalidades implementadas, sendo apenas necessário indicar alguns parâmetros que vão definir a estrutura da própria rede. A desvantagem neste tipo de APIs, é a grande abstração que as mesmas criam, o que pode ser uma desvantagem no caso do problema que estamos a tentar solucionar ser mais complexo.

```

return tf.estimator.DNNClassifier(hidden_units=hidden_layers,
                                   n_classes=2,
                                   feature_columns=features,
                                   activation_fn=activation_function,
                                   model_dir='/tmp/' + uuid.uuid4().hex,
                                   optimizer=tf.train.AdamOptimizer(learning_rate=learning),
                                   config=tf.contrib.learn.RunConfig(save_checkpoints_steps=250,
                                                                       save_checkpoints_secs=None,
                                                                       save_summary_steps=500))

```

Figura 3: Classificador

A figura 3, mostra o excerto de código referente ao estimador instanciada, que vai definir a estrutura da nova rede que vai ser criada.

Como é possível observar a nova rede vai ser criada com recurso a hiperparâmetros, que vão ser devolvidos pelo algoritmo de otimização descrito numa fase posterior deste relatório.

Os parâmetros definidos na rede são referentes ao número de camadas e nós, função de ativação e taxa de aprendizagem.

2.1 Modelo

```

training = tf.estimator.inputs.pandas_input_fn(x=x_train,y=y_train,batch_size=len(y_train), shuffle=True, num_epochs=3)
model = classifier.train(input_fn=training, steps=500)

```

Figura 4: Treino da rede

O modelo criado, vai ser treinado a partir do conjunto de dados fornecido, no entanto é necessário definir alguns parâmetros que permitem melhorar o processo de treino assim como o desempenho global do modelo. Parâmetros como o número de épocas a realizar, o tamanho do batch e o número de steps, são importantes, podendo resultar num modelo demasiado ajustado ao seu conjunto de treino, ou demasiado simplificado (**Overfitting**, **Underfitting**).

```

testing = tf.estimator.inputs.pandas_input_fn(x=x_test,batch_size=len(x_test),shuffle=False)
predictions = list(model.predict(input_fn=testing))

```

Figura 5: Teste da rede

O real comportamento do modelo vai ser verificado quando o mesmo for submetido a novos casos de teste. Neste caso a inexistência de um dataset específico para testes significa que o mesmo terá de ser validado recorrendo a um split do dataset, ou a um particionamento do mesmo em vários k-folds.

3 Validação

A validação do modelo de forma a verificar o seu comportamento quando exposto a novos casos de teste foi feita recorrendo a **TrainTestSplit** e a **K-Fold CrossValidation**

3.1 Particionamento Treino/Teste

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

Figura 6: TrainTestSplit

Este método consiste em dividir o dataset em 2 subconjuntos, face a inexistência de um conjunto próprio para validação. Embora seja possível validar o modelo de maneira simples, nem sempre os resultados obtidos são os mais próximos da realidade. A possibilidade de ocorrerem resultados demasiado otimistas, deve-se ao facto de que o subconjunto selecionado a partir do conjunto original, pode conter dados pouco representativos face a todo o conjunto. A vantagem no uso desta técnica é a sua rapidez, mostrando-se vantajoso quando existe a necessidade de validar múltiplos modelos.

A figura 6, refere o excerto de código referente aos subconjuntos originados de treino e teste gerados a partir do conjunto original, no qual o conjunto de testes vai ser composto por 30% do total de instâncias.

3.2 Validação Cruzada K-Fold

O uso de validação cruzada com k-folds, permite obter um comportamento mais realista do modelo. Neste caso vão ser realizados k testes por todo o dataset, possibilitando assim testar e treinar todas as instâncias.

```
def K_Fold_CrossValidation(dataset, K, randomise = True):  
  
    #Performing a shuffle, just before starting partitioning the dataset with k fold cross validation.  
    if randomise:  
        dataset = shuffle(dataset)  
  
    for k in range(K):  
        training = pd.DataFrame(columns=dataset.columns)  
        validation = pd.DataFrame(columns=dataset.columns)  
        for i in dataset.index:  
            if i % K != k:  
                training = training.append(dataset.iloc[[i]].copy(), ignore_index = True)  
            else:  
                validation = validation.append(dataset.iloc[[i]].copy(), ignore_index = True)  
  
        yield training, validation
```

Figura 7: Validação Cruzada

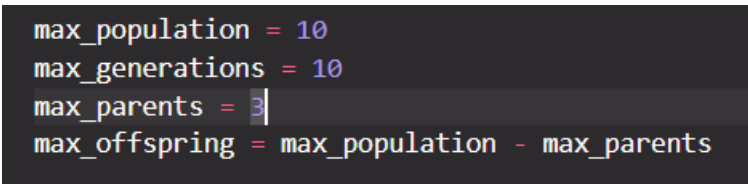
A figura 7, mostra um excerto do código referente à validação cruzada. Inicialmente é feito um shuffle ao dataset, impedindo assim que um dos conjuntos se diferencie em relação aos restantes.

Os k folds vão ser construídos, com base no cálculo do resto da divisão do index da instância pelo index do k fold atual, garantindo assim a construção em simultâneo do conjunto de treino e do conjunto de testes.

A desvantagem no uso desta técnica, é tempo mais demorado quando comparado com a técnica anterior, uma vez que vão existir k validações referentes a todo o dataset.

4 Algoritmos Genéticos

A implementação de algoritmos genéticos juntamente com redes neurais, tem como objetivo a otimização da própria rede, recorrendo à criação e validação de múltiplos modelos, que quando combinados geram um modelo ótimo, que se encontra mais adaptado ao nosso problema.



```
max_population = 10
max_generations = 10
max_parents = 3
max_offspring = max_population - max_parents
```

Figura 8: Parâmetros do algoritmo

A figura 8, mostra alguns dos parâmetros que foram definidos para o algoritmo genético. Definiu-se assim que a população vai ter um limite de 10 elementos, sendo renovada por um total de 10 gerações. O cruzamento vai ser feito entre os 3 pais com melhor desempenho nos seus modelos, dando origem a 8 novos descendentes.

A escolha do número de pais para fazer o cruzamento, assim como o número de descendentes gerados, pode levar a que a população fique estacionária, não apresentando qualquer evolução.

4.1 População

```
def Population():  
    global max_population  
  
    population = []  
  
    i = 0  
    while i < max_population:  
        chromosome = []  
        #Setting chromosome genes  
        chromosome.append(np.random.uniform(10**(-8), 10**(-2))) #Learning Rate  
        chromosome.append(np.random.randint(1,20)) #number of hidden layers  
        chromosome.append(2**np.random.randint(1,5)) #number of nodes per layer  
        chromosome.append(np.random.choice([0, 1])) #Activation function  
        population.append(chromosome)  
        i += 1  
  
    return population
```

Figura 9: População inicial

A população inicial vai ser constituída por um máximo de 10 elementos/cromossomas, cada um dos quais constituído por 4 genes. Cada um dos genes é um dos hiperparâmetros que vão ser passados para um novo modelo de rede neuronal
cada um dos cromossomas é constituído pelos seguintes genes:

- **Taxa de aprendizagem:** Varia entre 0,00000001 e 0,01
- **Camadas:** Varia entre 1 e 20
- **Nodos:** Pode variar entre 2 e 32
- **Função de ativação:** Varia entre a função sigmoid e relu

4.2 Aptidão

```
def Fitness(population, dataset):  
    global max_population  
  
    accuracys = []  
    i = 0  
    while i < max_population:  
        parameters = population[i]  
  
        accuracy = ANN.Initialization(False, dataset, parameters)  
        accuracys.append(accuracy)  
        i += 1  
  
    return accuracys
```

Figura 10: Função de aptidão

Na função de aptidão vão ser criados os vários modelos, recebendo cada um dos quais os hiperparâmetros referentes ao genes desse elemento da população. As accuracys dos vários modelos vão ser guardadas de modo a serem utilizadas numa fase posterior, para se determinar quais os elementos que tiveram melhor desempenho.

4.3 Pais

```
def Parents(population, fitness):  
    global max_parents  
  
    parents = []  
  
    i = 0  
    while i < max_parents:  
        max_fitness_idx = np.where(fitness == np.max(fitness))  
        max_fitness_idx = max_fitness_idx[0][0]  
  
        chromosome = []  
        chromosome.append(population[max_fitness_idx][0])  
        chromosome.append(population[max_fitness_idx][1])  
        chromosome.append(population[max_fitness_idx][2])  
        chromosome.append(population[max_fitness_idx][3])  
        parents.append(chromosome)  
  
        fitness = np.delete(fitness, max_fitness_idx, 0)  
        i += 1  
  
    return parents
```

Figura 11: Escolha dos pais

Em cada uma das gerações percorridas vão ser escolhidos os 3 pais mais aptos, ou seja, aqueles cujo os seus modelos obtiveram melhores desempenhos. Conforme referido anteriormente, o número de pais escolhidos como aptos deve ser ponderado, pois se existir muita discrepância no desempenho dos modelos, podemos cair no erro de escolher pais com desempenho relativamente fraco.

4.4 Cruzamento

```
def Crossover(parents):  
    global max_offspring  
  
    crossover_point = len(parents[0])/2  
  
    offsprings = []  
  
    i = 0  
    while i < max_offspring:  
        #selecting parents to crossover  
        parent1 = parents[i%len(parents)]  
        parent2 = parents[i%len(parents)]  
  
        offspring = []  
        for gene in range(len(parents[0])):  
            if gene <= crossover_point:  
                offspring.append(parent1[gene])  
            else:  
                offspring.append(parent2[gene])  
  
        offsprings.append(offspring)  
        i += 1  
    return offsprings
```

Figura 12: Função de Cruzamento

A figura 12, é um excerto do código referente ao cruzamento resultante dos pais mais aptos. Inicialmente é definido um ponto de corte, a partir do qual os genes de ambos os pais vão ser combinados. Neste caso definiu-se que o ponto de corte seria metade, ou seja, 2 genes da primeira parte do cromossoma do primeiro pai, com 2 genes da segunda parte do cromossoma do segundo pai.

A escolha dos pais para o cruzamento, tem por base o calculo referente ao resto da divisão do index atual do descente, pelo total de pais existentes na população.

Cada um dos novos descendentes vai ser guardado para numa fase superior receber algum tipo de mutação nos seus genes.

4.5 Mutação

```
def Mutation(crossover):  
  
    for i in range(len(crossover)):  
        #Generating a gene id to be mutated.  
        gene_idx = np.random.randint(0, len(crossover[0]))  
        #Saving current gene value  
        current_value = crossover[i][gene_idx]  
  
        while(True):  
            if gene_idx == 0:  
                new_value = np.random.uniform(10**(-8), 10**(-2))  
                if current_value != new_value:  
                    break  
            elif gene_idx == 1:  
                new_value = current_value + np.random.uniform(-1.0, 1.0, 1)  
                if new_value >= 1 and new_value <= 20:  
                    break  
            elif gene_idx == 2:  
                new_value = current_value + np.random.uniform(-1.0, 1.0, 1)  
                if new_value >= 2 and new_value <= 32:  
                    break  
            else:  
                new_value = 1 if current_value == 0 else 0  
                break  
  
        crossover[i][gene_idx] = new_value  
  
    return crossover
```

Figura 13: Função de Mutação

A função de mutação tem como objetivo garantir alguma diversificação na população existe, sendo aplicada aos descendentes gerados na geração atual.

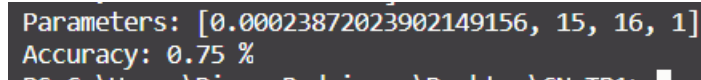
Conforme é possível observar, a mutação vai ser aplicada de forma aleatória num gene.

A mutação deve assim garantir que o valor atual seja modificado, e que não seja muito distante do valor atual. Uma grande discrepância nos valores de mutação, pode significar decréscimo na qualidade geral da população e de elementos futuros.

4.6 Paragem

A paragem do algoritmo genético é feito com base em 2 critérios, a primeira fase garante que existe um número de gerações a percorrer enquanto a solução a encontrar não for suficientemente satisfatória. A segunda fase permite interromper o ciclo de execução se a solução encontrada for superior a 80%, indicando assim que estamos perante uma boa solução.

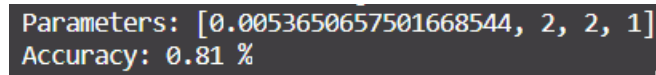
5 Resultados



```
Parameters: [0.00023872023902149156, 15, 16, 1]  
Accuracy: 0.75 %
```

Figura 14: Resultado com 10 gerações

A figura 14 mostra o resultado da execução com algoritmos genéticos após 10 gerações, ou seja, o algoritmo não conseguiu atingir o segundo critério de paragem de 80%, sendo obrigado a percorrer a totalidade de gerações. No entanto nesta execução em gerações anteriores, foi capaz de obter uma accuracy de 77%.



```
Parameters: [0.0053650657501668544, 2, 2, 1]  
Accuracy: 0.81 %
```

Figura 15: Resultado com 2 gerações

Na figura 15, é possível verificar o resultado quando o segundo critério de paragem foi atingido em apenas 2 gerações.

Em suma podemos observar que uma rede mais simples consegue melhores resultados quando comparada com uma rede mais complexa, definida pelos parâmetros da figura 14.

No entanto uma vez que o algoritmo só percorreu 2 gerações, não podemos concluir se ainda existia margem de manobra de nas próximas gerações a accuracy aumentar significativamente, no entanto se conhecermos bem o problema, podemos avaliar se o resultado obtido é suficiente ou não.

6 Conclusão

A realização deste trabalho prático mostrou a importância e a aplicabilidade de deep learning na sociedade atual, assim como a importância em criarmos uma rede que esteja adaptada ao problema que tentamos solucionar, daí a necessidade de realizar algum tipo de otimização na mesma.

Foi também possível ter uma percepção da necessidade de parametrizar corretamente os algoritmos genéticos, permitindo assim que os mesmos consigam chegar a uma solução aceitável, isto porque existe alguma dificuldade em definir a quantidade de pais aptos, descendentes e em definir o tipo de mutação a ser feito. Uma incorreta parametrização dos algoritmos genéticos podem levar a que a população fique demasiado estacionária sem qualquer melhoria nas aptidões obtidas.

A condição de paragem deve ser também bem ponderada, o ideal neste trabalho seria aplicar um critério de paragem, com a capacidade para verificar se existiu uma melhoria significativa, quando comparada com as duas gerações passadas. Isto é importante porque por vezes estamos próximos de uma solução ideal, não existindo espaço para uma melhoria significativa, e assim evitamos correr o algoritmo por mais gerações desnecessárias.

Como trabalho futuro seria interessante aplicar um modelo que seja mais sensível ao custo de errar, evitando ao máximo a existência de **True Negatives** e **False Positives**.

Aplicar otimização bayesiana de modo a comparar com os algoritmos genéticos seria um trabalho interessante.