Documentação Trabalho Prático III de Computação Natural: Classificando *Sloan Digital Sky Survey* usando Rede Neural

Ronald Davi Rodrigues Pereira
ronald.pereira@dcc.ufmg.br

Universidade Federal de Minas Gerais

04 de Dezembro de 2018

1 Introdução

Em algoritmos de agrupamento, uma aplicação muito recorrente é a de encontrar, separar dados n-dimensionais em grupos, de forma a juntar somente aqueles que forem similar, e aprender a predizer para dados futuros a qual grupo ele tem a maior probabilidade de pertencer. Existem vários algoritmos que se propõe a realizar essa tarefa, inclusive de aprendizado supervisionado, como por exemplo *k-Means*, e até mesmo de aprendizado não-supervisionado, como por exemplo as redes SOM (*Self-Organizing Map*).

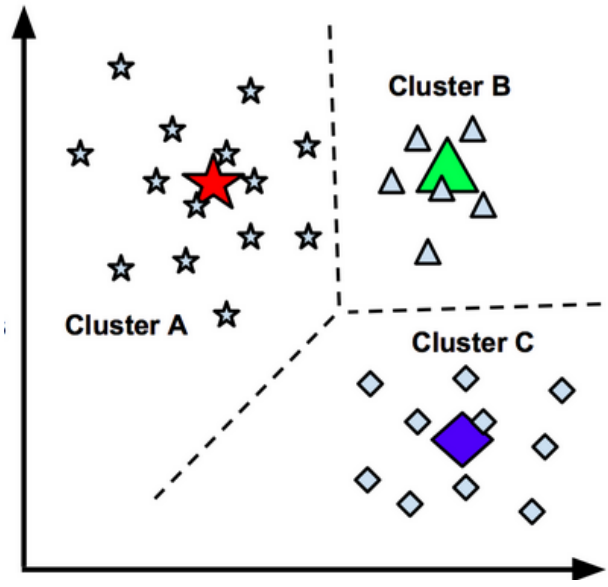

Figura 1 – Exemplo ilustrativo de um algoritmo de agrupamento

Como qualquer outro algoritmo de aprendizado de máquina, não é garantido que uma separação perfeita entre os grupos será alcançada, independentemente de quanto o algoritmo seja treinado. Porém, por meio de métricas, como por exemplo acurácia,

é possível saber o quão bem uma configuração de uma aplicação está desempenhando comparativamente.

O objetivo desse trabalho é de resolver o problema de classificação utilizando a base de dados do *Sloan Digital Sky Survey* utilizando uma rede neural. Portanto, buscamos uma solução que consiga maximizar o número de predições corretas para o grupo a quais dados em uma base de dados de testes pertencem. Para que seja possível alcançar tal feito com uma maior efetividade, vários métodos empíricos podem ser utilizados para tal, o que será discutido e mostrado nessa documentação para a base de dados proposta.

2 Modelagem e Implementação

2.1 Descrição dos Arquivos

sdss_prediction.py Programa principal que realiza o controle do fluxo de execução do programa

arg_parse_config.py Biblioteca customizada que realiza a configuração do *parsing* de argumentos para o programa principal

data_preprocessing.py Biblioteca customizada que realiza o pré-processamento dos dados

neural_network.py Biblioteca customizada que realiza a construção, compilação e execução da rede neural

requirements.txt Arquivo que explicita todas as bibliotecas externas (não-padrões do Python 3) utilizadas na confecção do trabalho, bem como as suas versões

input/ Pasta que contém todas as entradas

output/ Pasta que contém todos os testes executados

2.2 Estatísticas

Ao final de cada época, são impressas as seguintes estatísticas do conjunto de treinamento:

Epoch Época atual / Total de épocas

Time Tempo total para treinamento da época

Time per step Tempo gasto por cada *step* (execução de cada *batch*)

Loss Perda no conjunto de treino

Acc Acurácia no conjunto de treino

Val_loss Perda no conjunto de validação

Val_acc Acurácia no conjunto de validação

Como por exemplo:

```
Epoch 6/10
 - 5s 2ms/step - loss: 0.6147 - acc: 0.7860 - val_loss: 0.5325 - val_acc: 0.8273
```

Ao final da execução do treinamento (quando o número de épocas é alcançado), são impressas as seguintes estatísticas do conjunto de testes:

Global statistics Estatísticas globais

Loss Perda no conjunto de teste

Accuracy Acurácia no conjunto de teste

Class specific statistics Estatísticas para cada classe existente no conjunto de teste

Precision Precisão

Recall Revocação

F-Score Média harmônica entre precisão e revocação

Support Suporte (Quantidade de exemplos)

Confusion Matrix Matriz de confusão para o conjunto de teste

Como por exemplo:

```
------------------------------
Test set statistics
------------------------------
Global statistics
------------------------------
Loss: 0.5677
Accuracy: 0.8159
------------------------------
Class specific statistics
------------------------------
Precision for GALAXY: 0.7680
Precision for QSO: 0.0000
Precision for STAR: 0.9065
------------------------------
Recall for GALAXY: 0.9705
Recall for QSO: 0.0000
Recall for STAR: 0.7815
------------------------------
F-Score for GALAXY: 0.8574
F-Score for QSO: 0.0000
F-Score for STAR: 0.8394
------------------------------
Support for GALAXY: 880
Support for QSO: 138
Support for STAR: 682
------------------------------
```

```
Confusion Matrix
------------------------------
tr/pr   GALAXY  QSO     STAR
GALAXY  854     0       26
QSO     109     0       29
STAR    149     0       533
------------------------------
```

2.3 Construção da Rede Neural

Para a construção da rede neural foi utilizado o *framework* Keras[1] com a biblioteca Tensorflow[2] como *back-end*.

Para a modelagem da rede, foi escolhida um modelo de MLP (*Multi-Layer Perceptron*), em que consiste de camadas densas de neurônios conectadas por pesos e funções de ativação do sinal.

Na camada de entrada, foram utilizados 17 neurônios, de modo a representar cada dimensionalidade da entrada (excluindo a coluna *class*). Já nas camadas escondidas, o número de camadas e o número de neurônios em cada camada é configurável por passagem de argumento da linha de comando, sendo o padrão definido como 2 e 200, respectivamente. Esses números foram escolhidos a partir de resultados empíricos que tentaram balancear tempo de treinamento e acurácia.

Após cada camada escondida, foram utilizadas camadas de *dropout*[3] com valores randômicos definidos por uma função de probabilidade uniforme, de modo que, aleatóriamente, sinais de neurônios não eram propagados para a frente aleatóriamente, de modo a evitar *overfitting*[4] da rede e também para descobrir neurônios que não contribuem (ou até prejudicam) para a previsão da saída. Depois, foram utilizadas funções de ativações que são definidas aleatoriamente entre *sigmoid* (sigmoide) e *tanh* (tangente hiperbólica). Essas funções foram escolhidas pelo fato de serem boas funções para tarefas de classificação e muito utilizadas na literatura, além de que, nesse caso, não possuímos uma rede neural profunda por padrão, de modo que o problema de *vanishing gradient*[5] não aconteça.

Na camada de saída foram utilizados 3 neurônios apenas, de modo que cada um representasse uma classe distinta dada no conjunto de treinamento. Ao final da rede, após a camada de saída, foi utilizada uma camada que contém a função de ativação *Softmax*, que tem como objetivo retornar um vetor de 3 posições, em que cada posição indica uma classe, sendo a classe predita pelo modelo marcada com o número 1 e as demais como 0.

Para a função de perda, foi utilizada uma função de entropia cruzada categórica (*categorical cross-entropy*)[6], pois essa é a função de perda mais utilizada na literatura para problemas de classificação supervisionada em que se pode codificar todas os seus grupos em vetores de saída *one-hot*[7]. Já para a função de otimização e *backpropagation* foi utilizado o método do Gradiente Descendente Estocástico[8] (*Stochastic Gradient Descent: SGD*), que realiza a atualização dos pesos da rede conforme um certo número de exemplos (definidos pelo parâmetro *batch_size*) é passado pela rede e seus erros são calculados a partir de um algoritmo de minimização do erro baseado na Descida do Gradiente.

Para inicialização dos pesos, foi utilizada a função padrão do Keras, que inicializa todos os pesos de acordo com uma distribuição randômica uniforme. Vale ressaltar ainda que alguns parâmetros dessa rede são facilmente configuráveis por passagem de argumento: taxa de aprendizado (padrão: 0.001), número total de épocas (padrão 10) e tamanho dos

batches (padrão: 1).

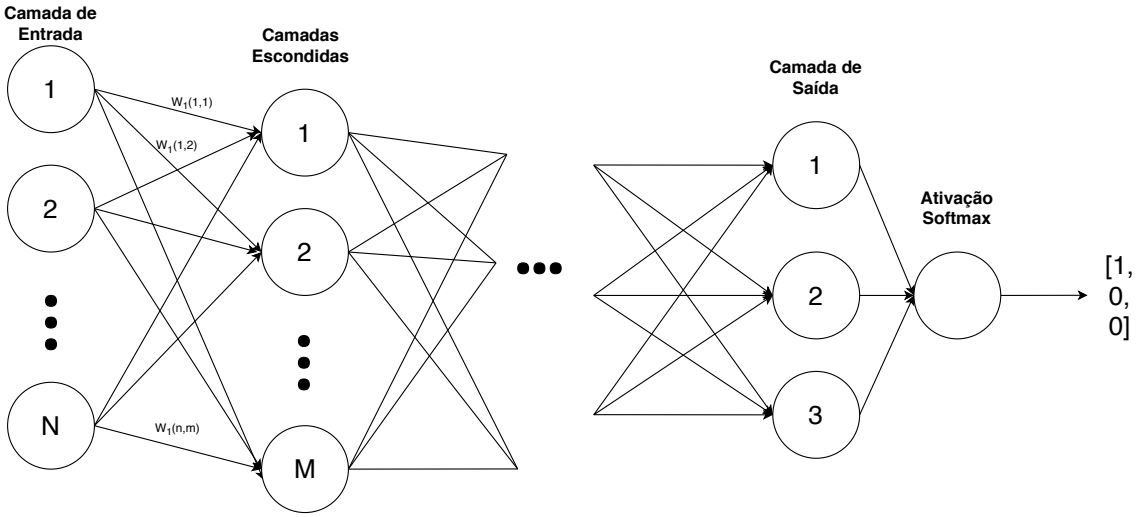

Figura 2 – Representação ilustrativa da rede *MLP* utilizada

2.4 Decisões de Implementação

Serão demonstradas abaixo todas as decisões de implementação que foram possíveis de serem tomadas durante o desenvolvimento desse algoritmo, de forma a explicitar quais estratégias foram utilizadas por mim nesse trabalho.

2.4.1 Partição Treino, Validação e Teste

Os dados de entrada foram particionados entre conjuntos de treino, validação e teste. Inicialmente, foi particionado aleatoriamente o conjunto de entrada, de modo que 66.6% pertença ao conjunto de treino e 33.3% pertença ao conjunto de teste. Já na etapa de treinamento, foi particionado aleatoriamente o conjunto de treino, de modo que 80% continue sendo conjunto de treino e 20% pertença ao conjunto de validação. Essas estratégias foram utilizadas de modo a evitar que a rede sofra um *overfitting* imperceptível, sendo que utilizando todos esses conjuntos de dados, caso isso aconteça, fica muito evidente, pela simples diferença latente entre a acurácia neles, e também para garantir que a rede consiga generalizar suas previsões para quaisquer tipos de dados futuros dessa base de dados.

2.4.2 Parâmetros e Execução

As variações de parâmetros foram implementadas de forma a facilitar testes e execuções distintas do algoritmo. Todos os parâmetros tidos como minimamente importantes foram externalizados para execução do programa principal com esses parâmetros passados por linha de comando.

Abaixo, um exemplo de uma execução com todos os parâmetros:

```
# Um menu de ajuda pode ser visualizado executando o comando:
# python3 sdss_prediction.py -h
python3 sdss_prediction.py -b 1 -e 10 -r 0.01 -l 2 -s 200 input/sdss.csv
```

-b Tamanho do batch para atualização dos pesos da rede pelo SGD (*Stochastic Gradient Descent*) (*integer* $\in [1, n)$, sendo n o número de instâncias no conjunto de treino). Padrão: 1

-e Total de épocas para o treinameneto da rede (*integer* $\in [0, \infty]$). Padrão: 10

-r Taxa de aprendizado do SGD (*Stochastic Gradient Descent*) (*float* $\in (0, \infty)$)). Padrão: 0.01

-l Quantidade de camadas escondidas (*integer* $\in [1, \infty]$). Padrão: 2

-s Quantidade de neurônios nas camadas escondidas (*integer* $\in [1, \infty)$) Padrão: 200

3 Experimentos e Resultados

Os experimentos abaixo foram executados repetidas vezes variando os parâmetros designados por cada seção abaixo. Esses testes foram executados em um computador com processador Intel Core I7 de 3.60GHz e 8 núcleos de processamento, com 16GB de memória DDR4 e sistema operacional Ubuntu 18.04.1 LTS. Além disso, foi utilizado a biblioteca do Tensorflow-GPU, de forma a executar os cálculos de maneira mais eficiente em uma placa de vídeo NVIDIA 960 GTX com capacidade de computação de 5.2[1].

Vale ressaltar que, como os testes teriam que ser variados, de forma a avaliar o desempenho da rede para diferentes configurações de conjuntos de dados, a semente para as funções *random* foi desativada, a fim de conseguir alcançar diferentes resultados para cada execução do algoritmo, sendo mantida somente na geração do modelo da rede neural para reprodução do modelo em diferentes execuções. Para executar o *script* interativo de execução de testes, basta executar os comandos abaixo:

```
cd src/
chmod +x tests.sh
./tests.sh
# Você será perguntado de quantas repetições do teste deseja executar
```

Todos os testes abaixo foram executados com os valores padrões, com exceção do parâmetro que está sendo variado, conforme descrito abaixo:

```
-b | --batch_size = 1
-e | --epochs = 10
-r | --learning_rate = 0.01
-l | --hidden_layers = 2
-s | --hidden_layers_size = 200
```

e foram realizados na base de dados *sdss.csv*.

[1] Esses valores podem ser conferidos no site oficial da NVIDIA no link https://developer.nvidia.com/cuda-gpus

3.1 Número de Camadas Escondidas

O número de camadas escondidas variou no conjunto {1, 2, 3, 4, 5}. Conforme o número aumentou, o tempo de execução do algoritmo e a qualidade das soluções cresceram um pouco, porém nada muito impactante em ambos, conforme pode ser observado pelo gráfico abaixo:

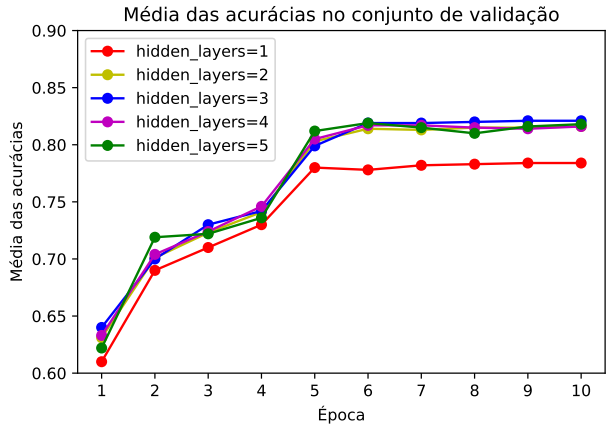

Figura 3 – Acurácia média no conjunto de validação para variação do número de camadas escondidas

No entanto, essa melhoria foi pequena, quando comparado aos outros parâmetros que foram variados. Portanto, o aumento do número de camadas escondidas além de 2 se mostrou desnecessário.

3.2 Número de Neurônios nas Camadas Escondidas

O número de neurônios nas camadas escondidas variou no conjunto {10, 50, 200, 500, 1000}. Conforme o número aumentou, o tempo de execução do algoritmo e a qualidade das soluções cresceram um pouco, porém, novamente, nada muito impactante em ambos, conforme pode ser observado pelo gráfico abaixo:

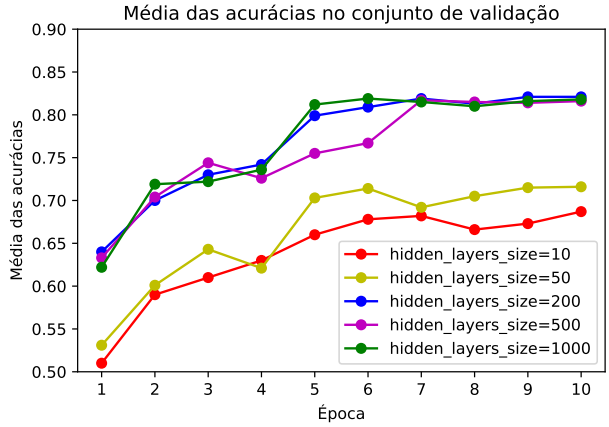

Figura 4 – Acurácia média no conjunto de validação para variação do número de neurônios nas camadas escondidas

No entanto, essa melhoria foi pequena, quando comparado aos outros parâmetros que foram variados. Concomitantemente, devido à alta variância provocada pela maior quantidade desse parâmetro, ele se torna uma medida complicadora para se convergir para uma solução candidata a ótima. Conforme também pôde ser observado, o número total de épocas para a execução desse aumento não foi muito necessário e um dos possíveis motivos para tal é a utilização das camadas de *Dropout*, de modo a desativar uma fração desses neurônios, diminuindo o número total de cálculos necessários.

3.3 Taxa de Aprendizado

A taxa de aprendizado variou no conjunto $\{0.001, 0.005, 0.01, 0.05, 0.1\}$. Conforme o número aumentou, o tempo de execução do algoritmo se manteve estável, porém a solução encontrada no final foi relativamente melhor quanto mais próximo de 0.01, conforme pode ser observado pelo gráfico abaixo:

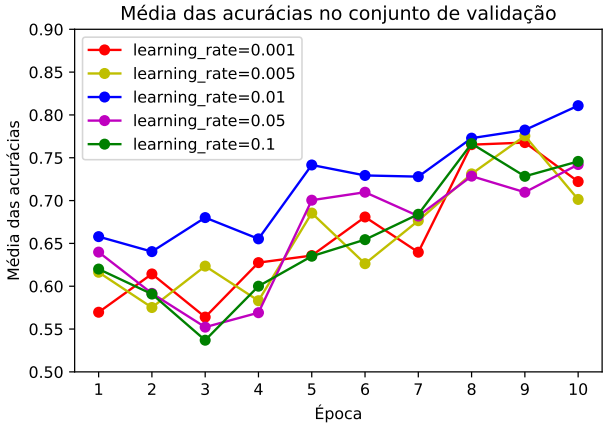

Figura 5 – Acurácia média no conjunto de validação para variação da taxa de aprendizado

Portanto, a variação desse parâmetro gera um comportamento na melhoria da solução que é facilmente observado. Desse modo, o número de iterações se mostrou um importante parâmetro a ser considerado. Para uma taxa de aprendizado que é ajustada ao longo das épocas de execução do treinamento, espera-se que, começando com 0.01, os resultados melhorem quanto mais próximo de 0.001 ele caminhe gradativamente.

3.4 Tamanho dos *Batches*

O tamanho dos *batches* variou no conjunto $\{1, 2, 5, 100, 500\}$. Conforme o número aumentou, o tempo de execução do algoritmo relativamente menor (pelo fato de menos execuções do *backpropagation*), porém a solução encontrada no final obteve resultados gradativamente piores, conforme pode ser observado pelo gráfico abaixo:

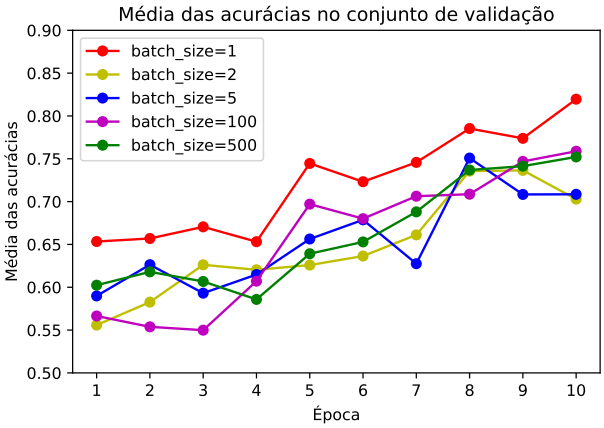

Figura 6 – Acurácia média no conjunto de validação para variação do tamanho dos *batches*

Esse teste demonstra também a execução do algoritmo de *backpropagation* do *SGD* a cada exemplo ($tamanho = 1$) e para *mini-batches* ($tamanhos > 1$). Esse resultado obtido pode ser explicado pelo fato de cada exemplo significar muito para a classificação do grupo, de modo que atualizar os pesos da rede conforme um peso mais generalizado (quando se aumenta o tamanho dos *batches*) diminui significativamente a qualidade da solução encontrada ao final, valendo mais a pena executar o SGD a cada exemplo da rede neural.

3.5 Total de Épocas

O total de épocas na fase de treinamento variou no conjunto {1, 2, 10, 20, 50}. Conforme o número aumentou, o tempo de execução do algoritmo ficou muito maior, porém a solução encontrada no final não se modificou muito a partir de 10 épocas, atingindo a convergência nesse ponto, conforme pode ser observado pelo gráfico abaixo:

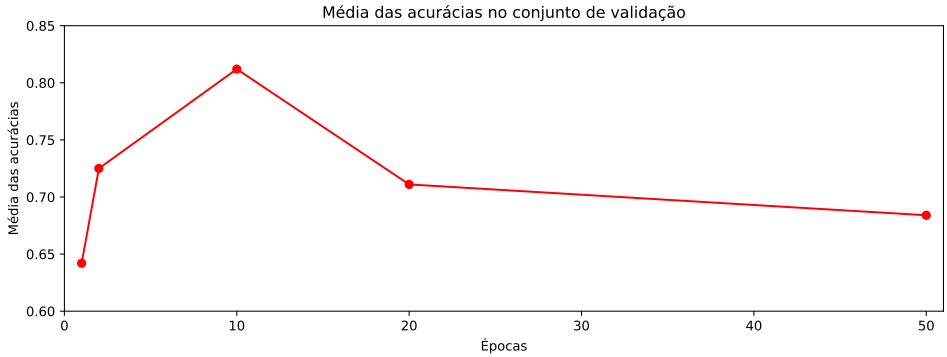

Figura 7 – Acurácia média no conjunto de validação para variação do total de épocas

3.6 Diferença da Acurácia entre Treino e Validação

Em geral, não foi obtido uma diferença muito grande entre as acurácias dos conjuntos de treino e validação. Isso pode ser explicado pelo fato do parâmetro *shuffle*

estar ligado (por padrão), de modo que ele aleatoriza esses conjuntos, evitando o *overfitting* do modelo.

3.7 Balanceamento de Classes

Foi implementada uma função para realizar o balanceamento das classes, de modo a replicar os exemplos da classe desejada na base de dados. Porém, tal balanceamento trouxe uma piora significativa para a acurácia do modelo, caindo de 0.8, em média, sem balanceamento para 0.5 com o balanceamento. Desse modo, percebe-se que, provavelmente, a classe 'QSO' se encontra próxima (ou até sobre-posicionada) das outras classes inicialmente majoritárias, dificultando o aprendizado do modelo de previsão. Portanto, o balanceamento de classes se mostrou prejudicial nesse trabalho.

3.8 Parâmetros Ótimos

Os parâmetros que se mostraram mais efetivos para a aproximação da solução ótima foram:

```
-b | --batch_size = 1
-e | --epochs = 10
-r | --learning_rate = 0.01
-l | --hidden_layers = 2
-s | --hidden_layers_size = 200
```

e, portanto, foram setados como parâmetros padrões na execução do programa principal. Desse modo, futuras execuções já poderão utilizar de parâmetros padrões otimizados anteriormente à partir dessas execuções experimentais anteriores.

4 Considerações Finais

Esse trabalho prático foi muito agregador no sentido de exercitar e procurar entender o quanto algumas variações de parâmetros impactam na acurácia de um modelo de previsão de redes neurais. Ele ajudou a fixar alguns conceitos de Redes Neurais, bem como tratamento dos dados de entrada, modelagem de uma rede neural *MLP* (*Multi-Layer Perceptron*), como representar uma solução categórica, quais camadas e funções de ativação utilizar, quais otimizadores performam melhor para certo problema de categorização, entre outros diversos tópicos importantíssimos para a área de Aprendizagem de Máquina, além de possibilitar um maior conhecimento prático nas bibliotecas *Keras* e *Tensorflow*.

Para tal, a confecção de testes variando esses parâmetros se mostrou uma tarefa exaustiva e demorada (pela repetições e tempo de finalização da execução de alguns testes), porém totalmente necessária para o entendimento do assunto proposto. No entanto, uma combinação exata desses parâmetros, de forma a prover a melhor acurácia para as previsões realizadas pelo modelo, ainda se mostrou de uma complexidade enorme, mas foi possível determinar certos subconjuntos de combinações que melhoravam relativamente essas execuções em busca dessas soluções.

Para a resolução de problemas de classificação por agrupamento, redes neurais emphMLP (*Multi-Layer Perceptron*) quase sempre possuem um ótimo desempenho quando comparado à outros demais métodos, incluindo várias outras técnicas de aprendizagem de

máquina. Isso demonstra o tamanho poder desses modelos para a resolução de problemas complexos em que são necessários diversos cálculos e estratégias distintas de se encontrar previsões com cada vez mais acurácia.

5 Referências Bibliográficas

[1] KERAS. Keras documentation. https://keras.io/. Acessado em 31 de Novembro de 2018.

[2] GOOGLE. Tensorflow documentation. https://www.tensorflow.org/. Acessado em 31 de Novembro de 2018.

[3] MLCHEATSHEET. Layers - dropout. https://ml-cheatsheet.readthedocs.io/en/latest/layers.html#dropout. Acessado em 31 de Dezembro de 2018.

[4] WIKIPEDIA. Overfitting. https://en.wikipedia.org/wiki/Overfitting. Acessado em 03 de Dezembro de 2018.

[5] WIKIPEDIA. Vanishing gradient problem. https://en.wikipedia.org/wiki/Vanishing_gradient_problem. Acessado em 02 de Dezembro de 2018.

[6] MLCHEATSHEET. Loss functions - cross entropy. https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy. Acessado em 31 de Dezembro de 2018.

[7] WIKIPEDIA. One hot. https://en.wikipedia.org/wiki/One-hot. Acessado em 02 de Dezembro de 2018.

[8] WIKIPEDIA. Stochastic gradient descent. https://en.wikipedia.org/wiki/Stochastic_gradient_descent. Acessado em 03 de Dezembro de 2018.