Processamento de imagens de Raio-X utilizando PySpark

Camila Lopes[1], **João Victor Nunes**[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)

{camila_ol,joaovictornunes}@id.uff.br

1. Introdução

Nesta seção serão descritos o objetivo, metodologia, métricas de comparação e limites do trabalho.

1.1. Objetivo

O objetivo do presente experimento é avaliar a efetividade de duas estratégias de processamento de um conjunto de dados específico, em um cenário de grande volume de dados, para se obter um modelo de Aprendizado de Máquina.

1.2. Metodologia

Para atingir o objetivo, foi escolhido um dataset de grande volume de dados de interesse dos autores. Depois, duas estratégias de processamento foram implementadas com a utilização da linguagem de programação *Python*. A primeira delas utilizou código *Python* puro. A segunda utilizou código Python com o framework *PySpark* [The Apache Software Foundation 2023], uma ferramenta otimizada para o processamento de grandes volumes de dados.

Nas duas abordagens, o objetivo final do processamento era a obtenção de um mesmo modelo de aprendizado de máquina, em um mesmo ambiente computacional.

1.3. Métricas de Comparação

A principal métrica de comparação utilizada foi o tempo de processamento. Dado o cenário de grande volume de dados, a velocidade é a principal métrica a ser avaliada. Outras métricas, como consumo de energia e de memória RAM, não foram utilizadas.

Além disso, foi utilizada a métrica de acurácia para avaliar o desempenho do modelo após seu treinamento.

2. *Dataset* e o Modelo de Aprendizado de Máquina

Nessa seção serão abordadas informações gerais do dataset.

2.1. Descrição dos Dados

O dataset utilizado é público e disponibilizado na plataforma Kaggle. É composto de 5.856 imagens validadas de Raio-X do tórax e está dividido em três categorias: pneumonia viral; pneumonia bacteriana; e normal [Kermany et al. 2018].

As imagens estão no formato .JPEG e divididas em imagens de treinamento, com 5.232 arquivos, e de teste, com 624 arquivos. O *dataset* possui tamanho total de 1.27 GB.

2.2. Preparação dos Dados

O pré-processamento dos dados seguiu as seguintes etapas: randomização e normalização dos pixels. Essas técnicas foram aplicadas de forma a evitar possíveis impactos advindos da ordenação do *dataset*.

2.3. Modelo de Aprendizado de Máquina

O modelo utilizado de Aprendizado de Máquina foi a Rede Neural Residual com 50 camadas ocultas, também conhecida como ResNet50. Essa rede neural de aprendizado de máquina profundo consiste na aplicação de camadas de convolução para a extração de diversas características (*features*) das imagens, com o objetivo final de classificá-las [He et al. 2016].

Para o trabalho, foi utilizada a ResNet50 pré-treinada com as imagens da competição ImageNet. A abordagem adotada, entretanto, não foi treinar a rede neural em si, mas usá-la para extrair e computar as features das imagens (etapa de *featurization*), que serão utilizadas posteriormente pelo classificador. A seguir, tem-se a etapa de *Label Encoding*, para conversão da coluna dos rótulos do tipo categórico para numérico para que o classificador consiga ser treinado efetivamente. Por fim, temos a etapa de predição em si, na qual o modelo de classificador final escolhido foi a Regressão Logística.

Além disso, o dataset é composto de três classes distintas. No entanto, para a implementação do modelo, foi considerado um problema binário, com as seguintes classes: pneumonia e normal. Ou seja, as classes pneumonia bacteriana e pneumonia viral foram agrupadas em uma somente.

3. Estratégias de Processamento

Nesta seção, serão abordados o ambiente de execução no qual as estratégias foram reproduzidas, assim como a descrição de cada estratégia.

3.1. Recursos Computacionais e Ambiente de Execução

O ambiente de execução utilizado foi o *Google Compute Engine* em *Python 3*, por meio da plataforma *Google Colab*. Neste ambiente, estavam disponíveis os seguintes recursos computacionais: memória RAM de 12.7 GB; um processador do modelo *Intel(R) Xeon(R) CPU @ 2.20GHz* com dois núcleos. Não houve a utilização de aceleradores.

3.2. Primeira Estratégia: *Python* Puro

A primeira estratégia adotada para a execução do experimento foi a utilização do *Python* Puro. Isto é, sem o uso de bibliotecas para aprimorar a execução do código. Neste caso, apesar de haver a disponibilidade de duas CPUs, as bibliotecas padrões do *Python* utilizam apenas uma.

Para esse código, foram utilizadas as bibliotecas Keras e Scikit-learn para o processamento dos dados de entrada e geração do modelo final.

3.3. Segunda Estratégia: *Python* com *PySpark*

Na segunda estratégia, foi utilizado o *framework PySpark*, criado para apoiar o processamento de grandes volumes de dados. Embora o Spark seja voltado para processamento

distribuído em um cluster com múltiplos nós, também é possível rodá-lo em um único nó (*standalone mode*). Nesse caso, os núcleos do processador podem ser usados como *workers*. No nosso ambiente de execução, portanto, contamos com um único nó, com dois workers, referentes aos dois núcleos do processador disponível.

Ainda dentro dessa estratégia, foi criada uma sub-estratégia, em que o código foi reexecutado, porém sem a reinicialização do ambiente de execução. Essa abordagem foi criada para avaliar o desempenho do sistema em um cenário em que um mesmo código é executado várias vezes sem a reinicialização do ambiente computacional, o que é uma prática comum em situações de criação de modelos de Aprendizado de Máquina, em que diversas configurações de hiperparâmetros precisam ser testadas.

Para essas execuções, utilizou-se majoritariamente a biblioteca ML do Spark, que substitui a antiga MLlib. O código é feito de modo a ser equivalente ao utilizado na execução com Python puro.

4. Resultados

Após a execução das diferentes estratégias, foram obtidos os resultados presentes nas tabelas 1 e 2.

Tabela 1. Tempos de processamento de cada estratégia

MaxIter	PySpark (1ª execução)	PySpark (reexecução)	Python Puro
10	13 min	40 s	12 min
100	19 min	6 min	14 min
1000	13 min	6 min	25 min
10000	13 min	6 min	65 min

Para a estratégia Python puro, o tempo de execução cresceu a medida que o número de iterações aumentou. Além disso, apesar de ter apresentado desempenhos melhores para os dois primeiros experimentos em relação à execução com PySpark (1ª Execução), para valores maiores de MaxIter, o tempo de execução tornou-se maior rapidamente.

Quando comparamos a execução de Python puro com o PySpark (reexecução), nota-se um desempenho bastante superior para a segunda estratégia, para qualquer valor de MaxIter.

Ao confrontar os tempos de execução das estratégias com PySpark, nota-se que há um ganho de desempenho significativo quando o código é reexecutado.

Em relação ao desempenho do modelo obtido ao final da execução de cada estratégia, medido pela acurácia, percebe-se que foram obtidos resultados iguais para ambas estratégias com PySpark. Com Python Puro, no entanto, o desempenho foi inferior para todos os valores de MaxIter.

5. Discussão e Conclusão

O artigo buscou avaliar o desempenho de duas estratégias de processamento de dados para realizar o treinamento de um modelo de Aprendizado de Máquina. Uma das estratégias

Tabela 2. Acurácia de cada estratégia

MaxIter	PySpark (1ª execução)	PySpark (re-execução)	Python Puro
10	0,86	0,86	0,82
100	0,86	0,86	0,81
1000	0,85	0,85	0,82
10000	0,85	0,85	0,82

foi realizada com bibliotecas padrões da linguagem de programação *Python*, enquanto a outra utilizou uma biblioteca denominada **PySpark**, voltada para o processamento de dados de grande volume.

O desempenho das estratégias foi avaliado por meio da aplicação em um conjunto de dados de Raio-X do tórax, cujo objetivo era classificar as imagens dentro das categorias ”pneumonia”e ”normal”. O modelo utilizado foi do tipo Rede Neural Residual, com 50 camadas ocultas, também conhecida como ResNet-50. Para o experimento, a rede neural utilizada havia sido pré-treinada com o conjunto de dados da competição ImageNet, de modo que ela foi responsável por extrair e computar as características das imagens.

Em relação aos resultados, foi percebida uma superioridade na velocidade de execução com o *PySpark* quando o número de iterações era grande (maior ou igual a 1000). Em relação à acurácia, a estratégia com o *PySpark* foi superior em todas instâncias do experimento.

Também foram realizadas reexecuções do código após a inicialização do ambiente de execução. Nesses casos, a velocidade de execução foi bastante superior à estratégia com Python Puro. Em relação à acurácia, foi exatamente a mesma que a primeira execução com *PySpark*. Os resultados nesse caso são justificados pelas otimizações no processamento presentes no framework.

Nesse sentido, a utilização do *PySpark*, mesmo para poucas iterações, mostrou-se útil devido ao superior desempenho do modelo obtido, além de aumentar bastante a velocidade de reexecução do código. Essa abordagem mostra-se, portanto, especialmente importante para aplicação em modelos de Aprendizado de Máquina, em que podem ocorrer muitas reexecuções do código.

Quanto a possíveis melhorias no experimento e oportunidades de trabalhos futuros, nota-se a necessidade de: uma avaliação estatística dos resultados obtidos; a realização do experimento para outros tipos de modelos de aprendizado de máquina; e a realização do experimento para variados tamanhos de conjunto de dados, com diferentes tipos de dados.

Referências

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Kermany, D., Zhang, K., and Goldbaum, M. (2018). *Labeled optical coherence tomography (oct) and chest x-ray images for classification*. Mendeley data 2.2 (2018): 651.

The Apache Software Foundation (2023). *Apache Spark*. https://spark.apache.org.