INF2102 - Projeto Final de Programação

Processamento de imagens de pneumonia em raio-x utilizando Deep Learning

Aluno: Luis Eduardo Craizer
Matrícula: 1912737

Orientador: Alberto Barbosa Raposo
Professor: Marcos Kalinowski

1. Introdução

Este projeto tem como objetivo construir uma rede neural que é capaz de interpretar imagens de radiografias de pulmões e prever a presença de pneumonia ou não nestas imagens.

A implementação do projeto é feita através da API Keras, em python, utilizando classes com orientação a objetos. Os modelos treinados no projeto são as redes neurais de convolução (ou ConvNet).

Para isto, são fornecidos duas redes neurais de convolução (ConvNets). Uma delas é um modelo convolucional simples, com camadas de ativação, criado do zero. O outro modelo com pesos e camadas iniciais já treinadas a partir da rede InceptionV3, utilizando a técnica de transfer learning. Ambos incorporam os fundamentos da programação orientada a objetos, simplificando assim o processo de desenvolvimento.

Os resultados geradas por esses modelos podem ser analisados através de uma matriz de confusão, cuja ideia principal é de oferecer uma avaliação mais completa do resultado, não apenas saber quais imagens foram corretamente rotuladas e quais não.

Em relação à interpretação destes resultados, idealmente queremos obter um número muito baixo de falsos positivos. Neste caso, é desejável que seja raro um paciente que esteja com pneumonia ser apontado como saudável pelo nosso modelo, algo que foi obtido por ambos os modelos.

2. Especificação do programa

Nesta sessão, será descrito o passo a passo da implementação do programa, apontando detalhadamente o que cada módulo faz e quais as suas respectivas funções na ordem cronológica seguida pela função principal do programa *main.py*.

2.1 Pré-configuração

O programa, para ser rodado, precisa da utilização da memória GPU. Para isto, nesta etapa, definimos um limite para a utilização da mesma.

Além disso, neste momento, é possível fazer o download dos dados, caso ainda não tenha sido feito. Os dados são então armazenados em na pasta *xray_data/*, contendo todos as imagens necessárias para a entrada do modelo.

Ambas as funcionalidades foram implementadas no módulo *preconfiguration.py*, localizado na pasta *preprocessing/*.

2.2 Carregamento dos dados

Nesta etapa, as imagens são carregadas a partir de suas respectivas pastas de armazenamento através do módulo *xray_loader.py*, na pasta *data_loader/*. Utilizando o pacote *numpy*, colocamos cada uma destas imagens em matrizes, separadas em 3 conjuntos:

- Treino;
- Validação;
- Teste.

Para construir o conjunto de validação, utilizamos a ferramenta do pacote *scikit-learn*, oferecendo como entrada os dados de treinamento. Tomamos uma proporção de 10% para os dados de validação, deixando o conjunto de treinamento com os 90% restantes destes dados. Além disso, a natureza da ocorrência das classes foi mantida ao declarar que os conjuntos sejam estratificados no momento da divisão.

Nesta etapa também, as imagens, cujas dimensões originais podem chegar a 1000, 2000 pixels de altura/largura, são redimensionadas para (150, 150, 3).

Com isso, temos o formato de cada matrix dado da seguinte maneira:

X_train: (4694, 150, 150, 3)
y_train: (4694, 2)

X_val: (522, 150, 150, 3)
y_val: (522, 2)

X_test: (624, 150, 150, 3)
y_test: (624, 2)

2.3 Visualizações prévias

Antes da rodar o nosso modelo, é possível ter um conhecimento maior dos dados. Através do módulo *preplotter.py*, presente na pasta *pre_visualization/*, geramos um *countplot* com a quantidade de imagens de radiologia de cada tipo: exames com pneumonia e exames sem pneumonia (normal).

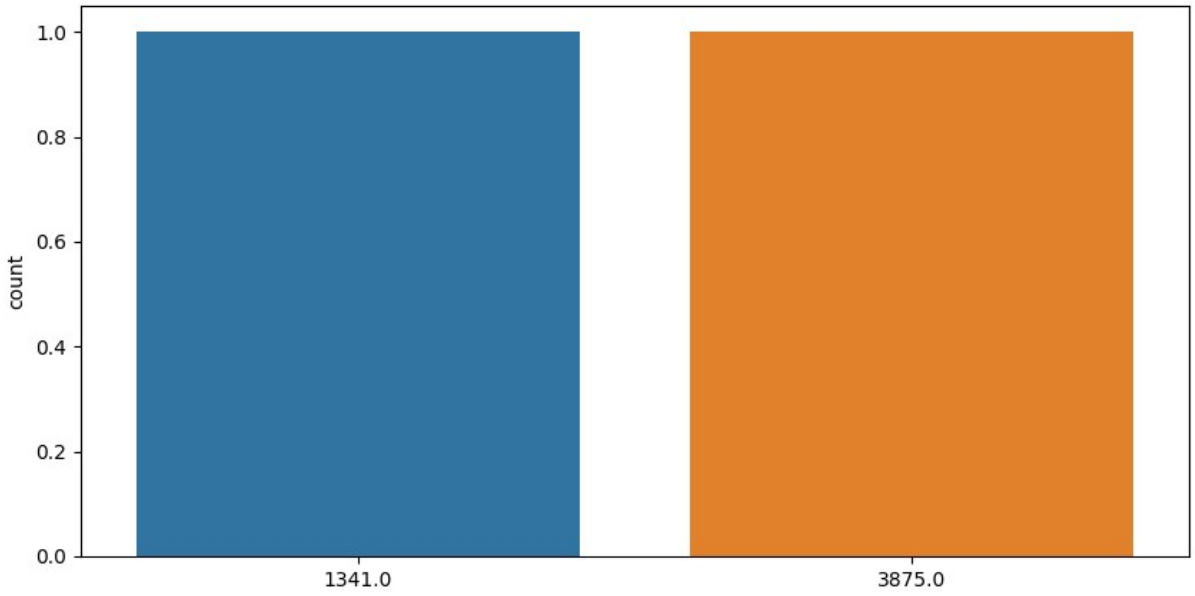


Figura 1: countplot. Em azul, quantidade de exames sem pneumonia; em laranja: com pneumonia

Neste momento de pré-treinamento, são gerados também exemplos de imagens de exames, como mostra a figura a seguir.

Figura 2: imagens de radiologia: à esquerda, exame que apresenta pneumonia; à direita, sem a doença

2.4 Treinamento do modelo

Chegamos finalmente na etapa de treinamento do modelo. Como já dito anteriormente, foram treinados dois modelos convolucionais: um modelo simples criado do zero, com múltiplas camadas - descrito no módulo *conv_model.py* e um modelo que utiliza a abordagem de Transfer Learning e utiliza os pesos e as camadas de uma rede já treinada (Inceptionv3), disponível pelo pacote Keras - descrito no módulo *TL_model.py* (localizados na pasta *model/*).

Em ambos os módulos, o processo é o mesmo:

1) Inicialização do modelo;
2) Criação do corpo de modelo;
3) Compilação do modelo;
4) Treinamento do modelo.

Na etapa 1, o que difere os modelos é que no primeiro modelo é inicializado um modelo sequencial, pronto para receber as camadas que vão dar o corpo ao modelo. Já no caso do segundo modelo, sua inicialização se dá carregando o modelo base InceptionV3, com pesos e camadas já fixados.

Na segunda etapa, são construídas as camadas dos modelos. Para o modelo convolucional simples, são construídos blocos compostos por: camada de convolução, camada de normalização (utilizando *BatchNormalization*), camada de ativação (*relu*) e *MaxPooling2D*. À medida que a dimensão do altura e largura do bloco vai se encolhendo, a quantidade de filtros aumenta (indo de 16 até 128 na camada de convolução), estratégia bastante utilizada em redes neurais. Nas últimas camadas, adicionamos mais uma camada densa, seguido por um Dropout (reduzindo o overfitting) e uma camada densa com ativação *sigmoid*, que ativa o neurônio, entre duas possibilidades, com a maior propensão de estar correto.

Em relação ao segundo modelo sua construção se dá basicamente por fundir as camadas vindas da rede já treinada InceptionV3 com as camadas finais de *Dropout*, *GlobalAveragePooling*, *Dense* e *BatchNormalization* e, finalmente, uma última camada densa com ativação *sigmoid*, também responsável por ativar o neurônio com maior probabilidade.

As últimas duas etapas não diferem em nada em termos de implementação. A uńica diferença está nos parâmetros utilizados para compilar cada um. No modelo simples *ConvolutionalModel*, a função de otimização utilizada é o *RMSProp*, enquanto no *TransferLearningModel*, utilizamos o *Adam*.

Foram utilizados também dois hiperparâmetros comuns na literatura das redes neurais: os callbacks *ReduceLROnPlateau*, responsável pela redução gradual da taxa de aprendizado numa alcançar melhores resultados e *ModelCheckpoint*, que salva a configuração de pesos e ativações referentes à época de melhor aprendizado do modelo de acordo com algum critério (no nosso caso, de acordo com a *validation_loss*).

2.5 Fase de teste

Depois de treinamento, a fase de teste é o momento em que o modelo é avaliado. A função *evaluate_model*, também presente em cada um dos módulos referentes aos diferentes modelos realiza esta etapa.

Depois de treinado, chamamos a função *model.predict* do pacote Keras, dando como entrada o conjunto de teste que foi carregado na etapa 2.2. Rodamos finalmente esta função, que classifica cada um das 624 imagens do conjunto em normal ou pneumática, de acordo com o que foi aprendido na etapa anterior.

2.6 Visualizações finais

Finalmente, o módulo *plotter.py* é responsável por gerar as visualizações finais. Uma delas é o histórico do treinamento, ilustrado através de um gráfico de *training/validation loss* por época de treinamento.

Os resultados dos testes podem visualizados através de uma matriz de confusão, que ilustra como se comportam as predições do modelo. A figura a seguir ilustra o funcionamento da matriz de confusão.

		Detectada	
		Sim	**Não**
Real	**Sim**	Verdadeiro Positivo (VP)	Falso Negativo (FN)
	Não	Falso Positivo (FP)	Verdadeiro Negativo (VN)

Figura 3: Matriz de confusão

Idealmente, queremos atingir uma alta porcentagem de verdadeiros positivos e de verdadeiros negativos, ou seja, desejamos que o modelo faça previsões corretas. Desta forma, queremos alcançar um número baixo de falso negativos e, mais importante ainda, um número baixíssimo de falsos positivos, uma vez que são estes pacientes que são ditos saudáveis e que, na verdade, eram portadores da doença.

3. Estrutura do projeto

3.1 Diagrama de sequência

O diagrama de sequência é representado na Figura 4, descrevendo todo o programa do início ao fim. O início se dá com a leitura de algum dado/informação de entrada e termina com o resultado final do processo.

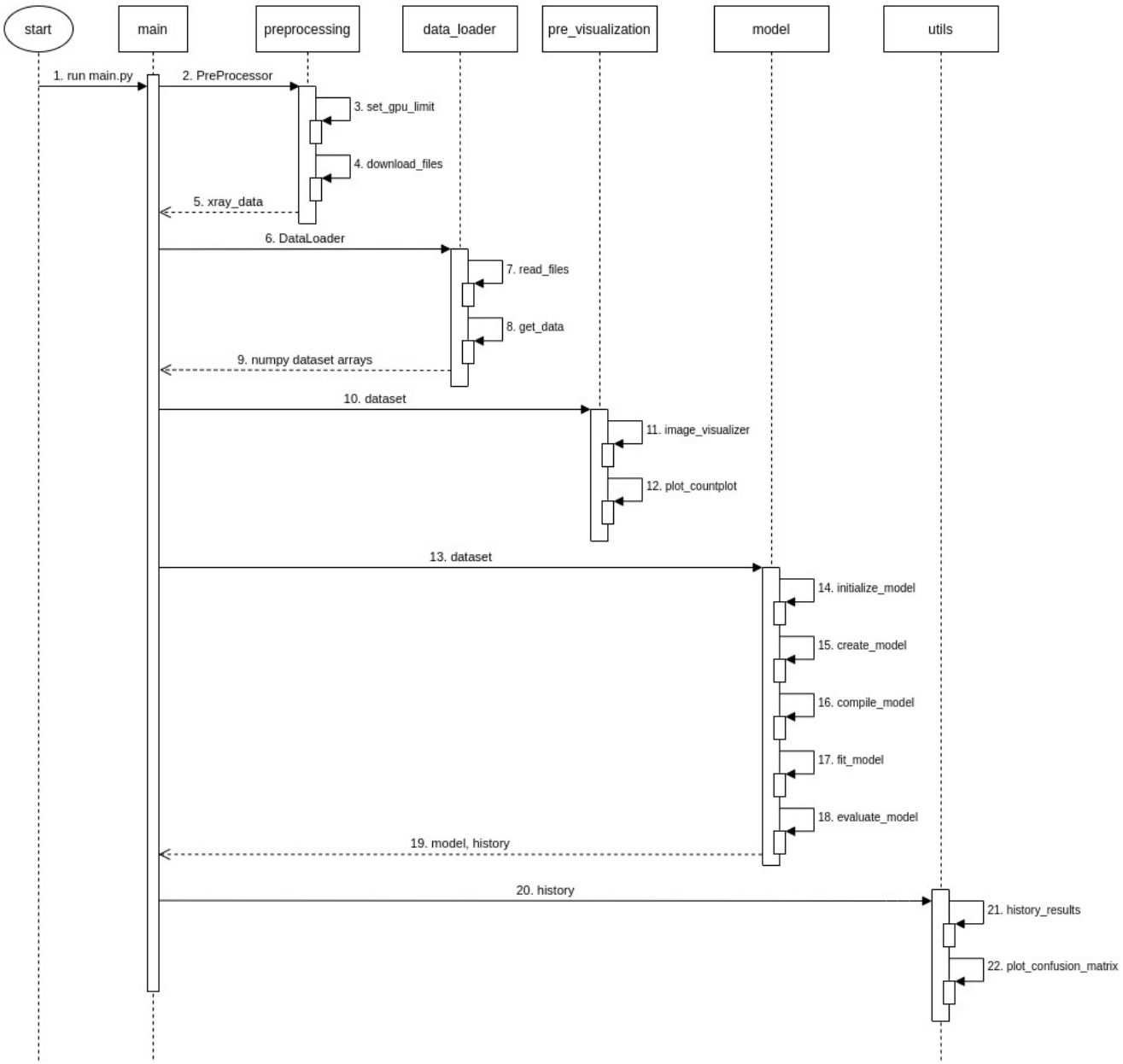


Figura 4: Diagrama de sequência

3.2 Diagrama de classes

O diagrama de classes é ilustrado na Figura 5, representando os módulos do programa e as relações entre as classes. Neste experimento, foram utilizadas as seguintes classes:

- **PreProcessor:** configura um limite de uso da GPU e baixa os arquivos de imagens.
- **DataLoader:** carrega as imagens e as salva em matrizes *numpy.* Separa os dados em conjuntos de treino, validação e teste.
- **PreVisualizer:** fornece informações sobre os dados antes da etapa da treinamento.
- **Model:** inicializa, cria, treino e avalia o modelo. Classe presente nos módulos *conv_model* e *TL_model*, variando apenas o nome do modelo.
- **Visualizer:** gera visualições do treinamento do modelo e dos resultados obtidos através de uma matriz de confusão.
- **Main:** faz a chamada de todos os módulos e inicia o programa

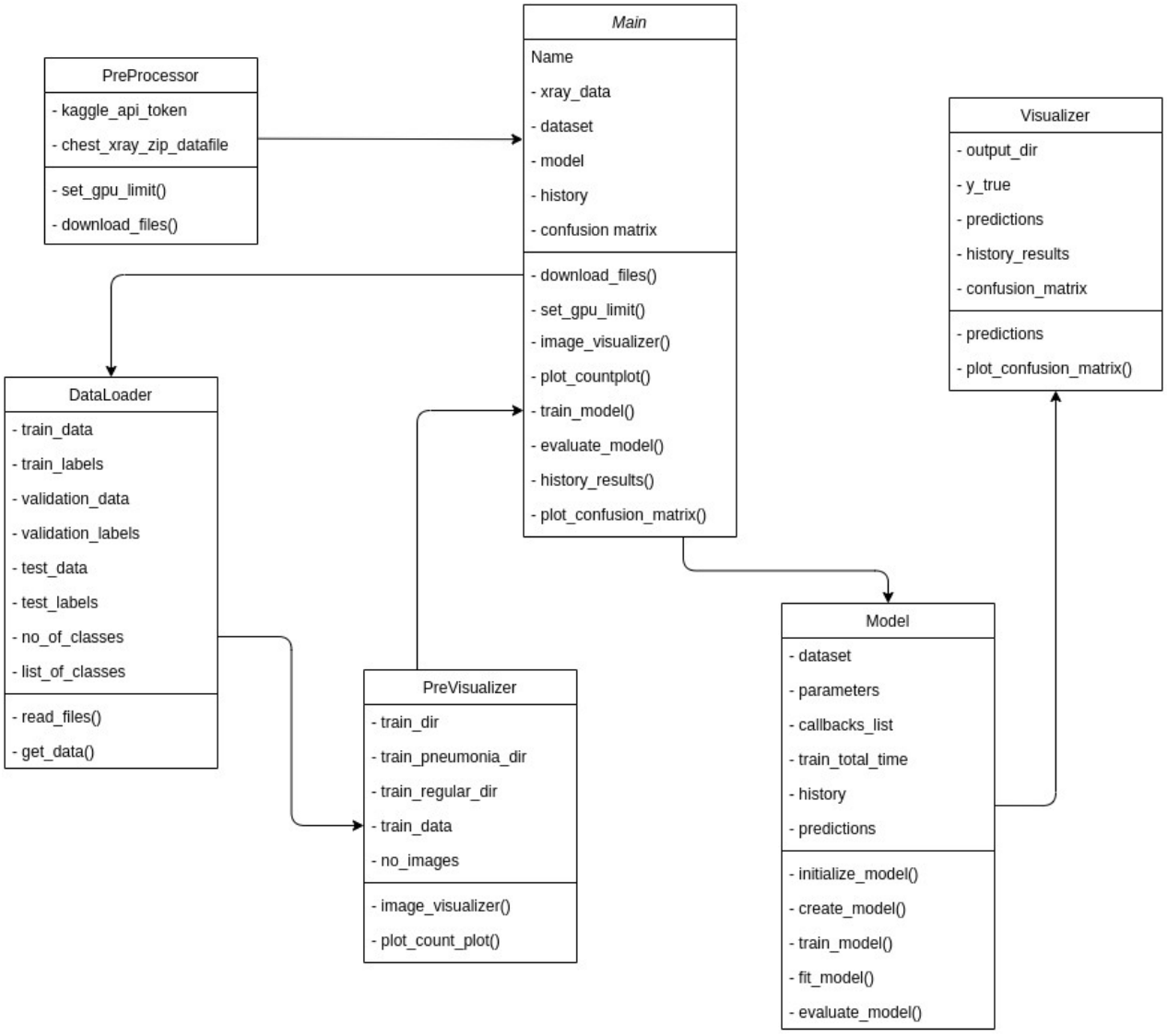


Figura 5: Diagrama de classes

4. Resultados

Esta sessão contém os resultados obtidos pelos modelos. Primeiro, temos resultados sobre o treinamento dos mesmos e, na sequência, os resultados obtidos pela predição das classes do conjunto de teste.

4.1 Resultados de treinamentos

- Conv_model: como podemos observar, o comportamento do modelo simples convolucional foi de altas variações em sua *validation_loss* até a sexta epóca. Depois disso, o modelo parece ter convergido a um valor baixo de *validation_loss*, terminando o treinamento na 9ª época, o que é um sinal de que o treinamento pode ter sido rápido e eficaz.

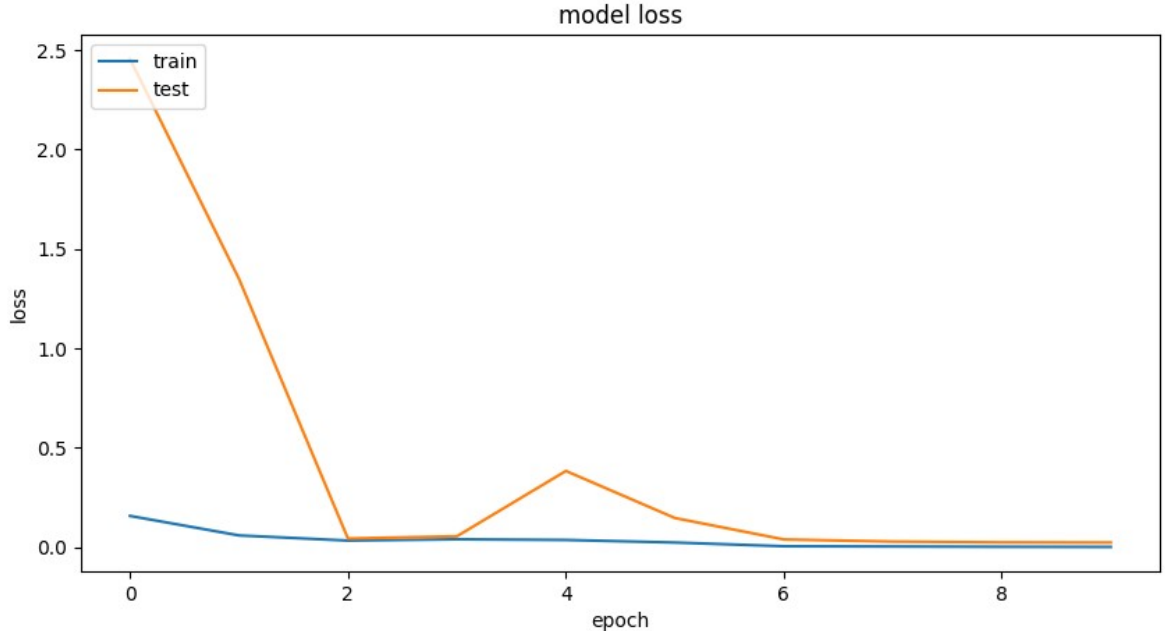


Figura 6: Gráfico do treinamento do *conv_model*

- TL_model: o treinamento deste modelo se deu de forma parecida com o do outro modelo, tendo também sua *validation_loss* mudando pouco a partir da sexta época. Apesar de poucas variações a partir desta época, o treinamento rodou até a época 18.

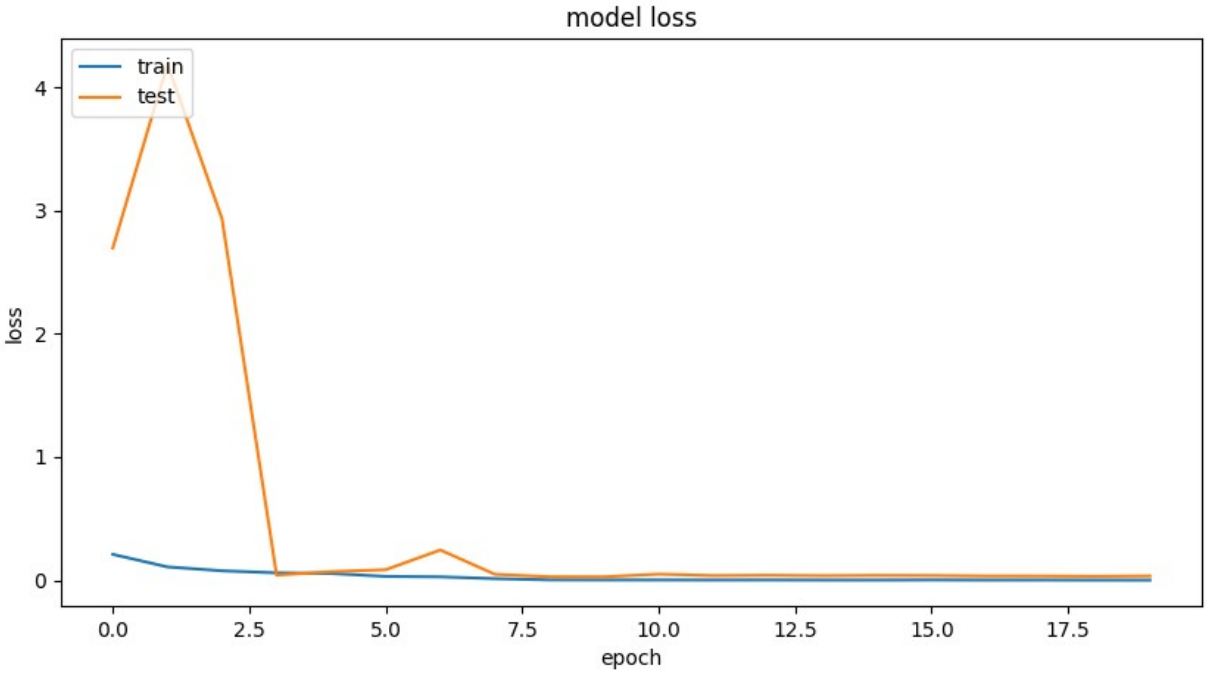
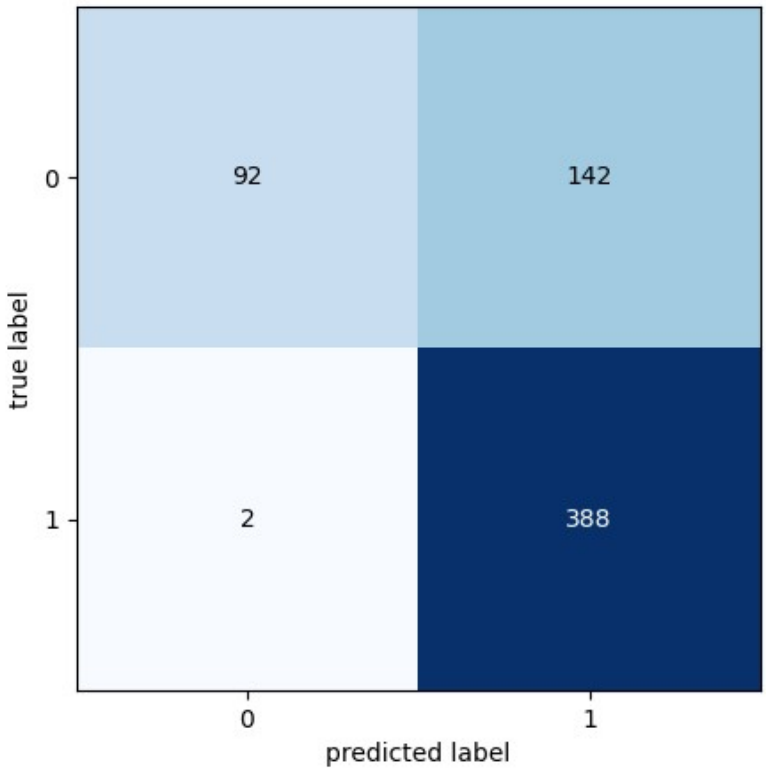


Figura 7: Gráfico do treinamento do *TL_model*

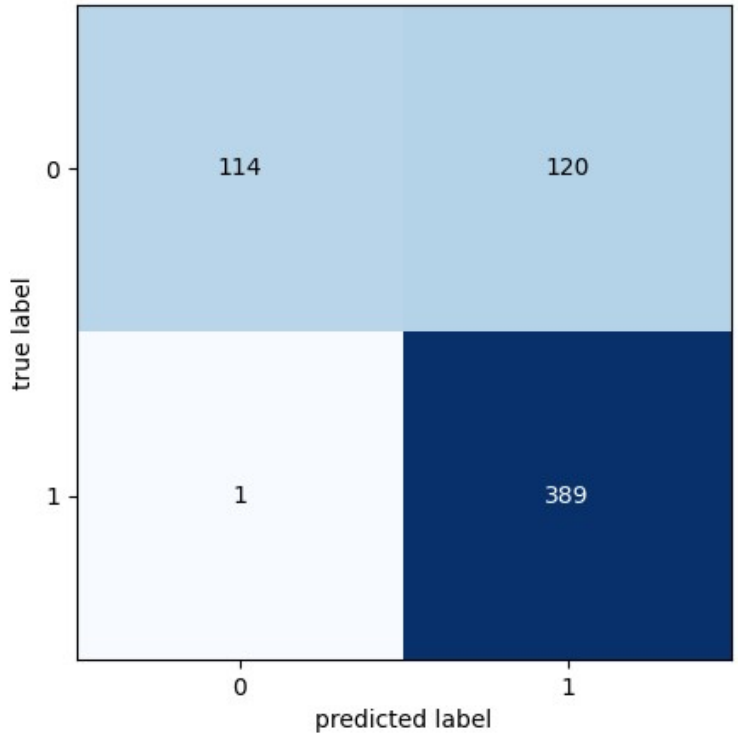
4.2 Resultados de testes dos modelos

Uma vez os modelos treinados e avaliados no conjunto de teste, temos os resultados ilustrados a seguir através da matriz de confusão.

- **Conv_model:** mostrou-se um resultado satisfatório do ponto de vista de falsos negativos - apenas 2 classificados. Já sobre o olhar de falsos positivos, o modelo chegou a valor alto - 142 imagens, o que não é tão satisfatório.



TL_model: como esperado, este modelo chegou a resultados um pouco melhores que o outro. Foi alcançado apenas 1 falso negativo, porém ainda obteve-se u valor alto de falsos positivos – 120 encontrados.



5. Conclusões

Convnets são redes com enorme potencial em relação à classificação de imagens. Já existem estudos que comprovam que é possível, em diversas áreas, alcançar resultados melhores do que os obtidos por análise de humanos.

O trabalho utiliza a abordagem de classes orientadas a objeto, tendo assim toda sua estrutura organizada de maneira a facilitar a sua implementação e a sua compreensão.

Os resultados obtidos foram consistentes com a expectativa: baixo valor de falso positivos, que é o aspecto mais relevante do estudo. Apesar disso, a quantidade de falso positivos encontrados é um número que pode ser reduzido através de novos estudos sobre o problema. Mesmo sabendo que sua gravidade não é tão alta quanto um possível alto resultado de falsos negativos, é um número que buscamos sempre que seja o menor possível.

A aquisição de dados na área médica pode ser extremamentea rara e custosa. Desta forma, nem sempre é possível utilizar aprendizado supervisionado para o treinamento de modelos. Em outras palavras, nem sempre é possível alimentar o modelo com os rótulos das possíveis classes sobre cada imagem, para que o mesmo saiba identificar a qual grupo pertence cada imagem.

Deixamos como sugestão para futuras pesquisas a utilização da técnica de detecção de anomalias, que se apresenta como uma alternativa a esse problema. Com ela, é possível treinar um modelo que é capaz de prever quais são os dados considerados normais e os considerados anômalos (neste caso de uso, pneumáticos) sem a necessidade de rotular cada imagem para alimentar o modelo.