

Desafio 2 - Redes neurais convolucionais para distinguir cães de gatos

Trabalho de Grupo realizado no âmbito da Unidade Curricular
de Aprendizagem Profunda para Visão por Computador do 1º
ano do Mestrado em Ciência de Dados

Diogo Freitas, 104841, MCD-LCD-A1

Diogo_Alexandre_Freitas@iscte-iul.pt

João Francisco Botas, 104782, MCD-LCD-A1

Joao_Botas@iscte-iul.pt

Miguel Gonçalves, 105944, MCD-LCD-A1

Miguel_Goncalves_Pereira@iscte-iul.pt

Ricardo Galvão, 105285, MCD-LCD-A1

Araujo_Galvao@iscte-iul.pt

30 de março 2025

Versão 1.0.0

Índice

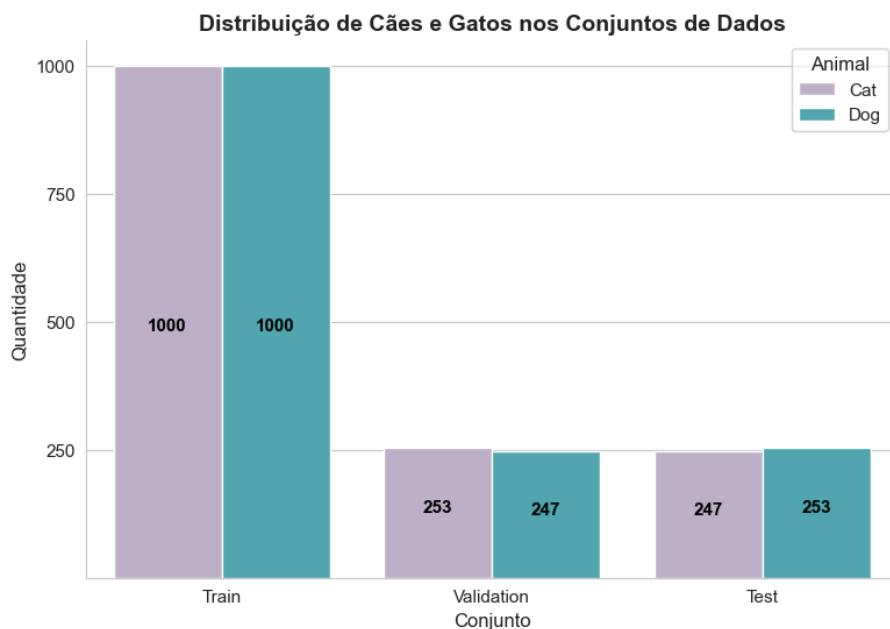
Parte 1 : Preparação dos dados

Para a realização deste trabalho, foi utilizado o dataset [Dogs vs. Cats](#) do Kaggle, com o objetivo de classificar imagens de cães e gatos, abrangendo uma ampla diversidade de raças e géneros. O conjunto de dados apresenta desafios adicionais, como a ausência de um padrão definido, a presença de ruído, variações na orientação, diferenças na iluminação e inversões, fatores que podem dificultar a tarefa de classificação por redes neurais convolucionais (CNNs). Além disso, as imagens possuem dimensões variadas e, para garantir um formato padronizado, é realizada uma operação de redimensionamento durante a importação, utilizando interpolação bilinear¹, para assegurar que todas as imagens fiquem com a dimensão de 256×256 pixels. A [Figura 1](#) apresenta dois exemplos que podem dificultar a capacidade preditiva dos modelos.



Figura 1: Exemplo de imagens do conjunto de dados - com orientação diferente à esquerda e com ruído e divergência de luminosidade à direita

As imagens foram fornecidas em 2 pastas/conjuntos: um de treino com 1000 imagens, distribuídas equitativamente entre cães e gatos, e outro de validação também com 1000 imagens, que continham 500 gatos e 500 cães. O conjunto de validação foi dividido em dois para criar um conjunto de teste² com o “mesmo” tamanho do de validação, cuja distribuição das classes pode ser vista na [Figura 2](#).



Nota-se que a distribuição do conjunto de validação/teste não é exatamente 50/50 para as duas classes, pois depende do caráter pseudo-aleatório da seed fornecida na divisão da função do keras. A seed fixada foi 777, neste caso.

Figura 2: Distribuição de cães e gatos nos conjuntos de dados

Parte 2 : Rede Neuronal convolucional “custom”

Nesta fase do desafio vamos utilizar uma rede convolucional “custom”, ou seja, com camadas convolucionais e *layers* de *pooling* definidas e construídas por nós. Para este efeito, dividimos o treino em 5 modelos com particularidades diferentes, a fim de identificar possíveis melhorias entre experiências e a fazer *tuning* das *features* da nossa rede.

¹https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory

²https://keras.io/api/data_loading/image/

Inicialmente, como modelo de referência (*baseline model*), treinamos uma rede neuronal com duas camadas convolucionais: a primeira com 16 filtros de tamanho 5×5 e a segunda com 32 filtros de tamanho 3×3 . Após isso, aplicamos uma camada de *max pooling*, seguida de uma terceira camada convolucional com 64 filtros de tamanho 3×3 e outra camada de *max pooling* para reduzir ainda mais a dimensionalidade e chegar com a informação mais relevante ao final da rede. Para finalizar transformamos tudo num vetor 1D com *Flatten* e utilizamos 2 camadas densas: uma com 128 neurónios e ativação ReLU, seguida da camada de saída com 1 neurónio e ativação sigmoide para o nosso problema de classificação binária (cão/gato).

A partir deste modelo *baseline* (1) foram feitos incrementos a este modelo em mais 4 fases do treino, sendo estas:

- (2) **Modelo com mais 1 layer convolucional:** Foi adicionada uma camada com 128 filtros de tamanho 3×3 logo após a camada com 64 filtros, antes do 2º *max pooling*. A inserção desta camada foi sobretudo para aumentar a robustez e diversidade da rede e tentar filtrar a informação relevante;
- (3) **Adicionado Dropout:** Foi adicionada uma camada de *Dropout* com uma taxa de 30% antes da camada de *output*, para tentar reduzir/evitar *overfitting*;
- (4) **Data Augmentation (DA):** Como já foi mencionado anteriormente (Figura 1), as imagens continham propriedades diferentes no que toca a luminosidade, orientação, entre outras. Por esta razão decidimos fazer DA com os seguintes “*augments*”: inversão horizontal, rotação aleatória, zoom in/out, ajuste de contraste e ajuste de brilho, todos com variações de até 10%;
- (5) **Batch Normalization (BN):** O BN serve essencialmente para reduzir a variação na distribuição das ativações das camadas durante a fase de treino (*internal covariate shift*). Foi utilizado como uma tentativa de acelerar a convergência e melhorar a generalização do modelo, assim procurando mitigar o risco de *overfitting*.

Para se entender melhor as etapas da fase de treino mencionadas, conseguimos observar a arquitetura base da nossa rede, na Figura 3. A montagem desta arquitetura baseia-se no *baseline* e tem os incrementos de forma explícita com (N).

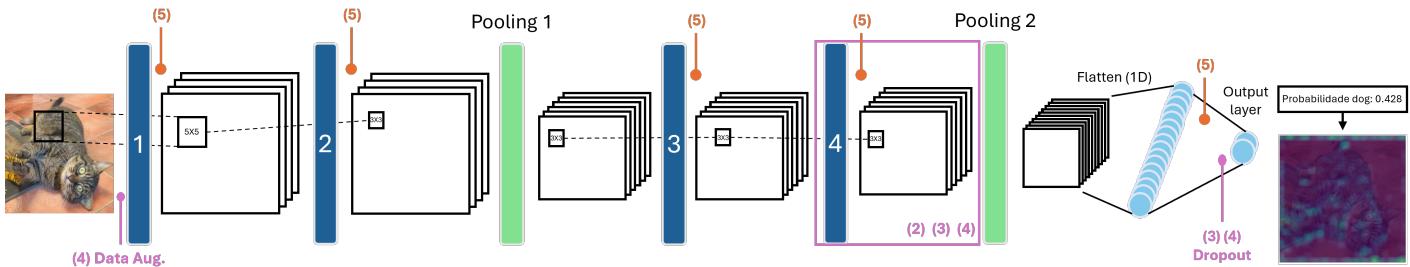


Figura 3: Arquitetura da rede “custom” criada

(2) - Mais uma *layer convolucional*, (3) - Adição *Dropout*, (4) - *Data Augmentation*, (5) - *Batch Normalization*

Para as 5 fases, a rede foi compilada com o otimizador *Adam*, com um *learning_rate* de 0.001; com a função de perda *binary_crossentropy*, ideal para problemas de classificação binária (cão ou gato); e por métricas, entre elas a *accuracy*, *precision* e *recall*. Posteriormente, as 5 redes foram treinadas para 50 *epochs* e com 3 *callbacks*, entre elas:

- **BEST_MODEL_CHECKPOINT:** guarda os pesos do modelo com menor valor da função de perda;
- **EARLY_STOPPING:** interrompe o treino quando não há melhorias na função de perda num período de 5 épocas;
- **REDUCE_LR:** reduz o *learning rate* quando a função de perda na validação não melhora durante 7 épocas consecutivas. Caso não haja melhoria, é reduzido para 50% do valor atual, mas não pode atingir um valor mínimo de 1×10^{-6} .

Com o treino efetuado, foram ainda gerados mapas de saliência com o intuito de analisar quais regiões da imagem o modelo considerou mais relevantes ao tomar as decisões de classificação. Esta análise é crucial para entender possíveis falhas na interpretação do modelo, pois, em alguns casos, a atenção pode estar mais focada em áreas irrelevantes ou enganadoras, como fundos “ruidosos” ou padrões aleatórios, em vez das características distintivas do objeto principal.

Tabela 1: Resultados dos modelos com o treino da rede convolucional (*training on 2 components*)

Modelo	Tempo de Execução		Nº Epochs	Métricas de Desempenho			Nº Parâmetros	
	GPU ³	CPU ⁴		Precision	Accuracy	Recall	Treináveis	Totais
Baseline_model	29.2s	3m 44s	6	0.648	0.658	0.656	8,413,217	25,239,653
+ 1 layer conv2D()	37.9s	5m 22s	7	0.429	0.592	0.627	16,875,681	50,627,045
+ Dropout	52.5s	5m 50s	9	0.696	0.632	0.612	16,875,681	50,627,045
+ Data augmentation	2m 23.8s	22m 57s	25	0.753	0.714	0.694	16,875,681	50,627,045
+ Batch normalization	4m 16.4s	32m	19	0.789	0.728	0.699	16,876,417	50,629,989

Pela Tabela 1, observamos que o melhor modelo é o que utiliza BN, em que alcança as melhores métricas de desempenho. Destaca-se também o impacto positivo do DA, que melhora significativamente a performance, talvez devido à correção da diversidade das imagens. Além disso, os tempos de execução na GPU são consideravelmente menores que na CPU, reforçando a eficiência do uso de aceleração por *hardware*. Analisaremos o modelo final (5) com BN em mais detalhes:

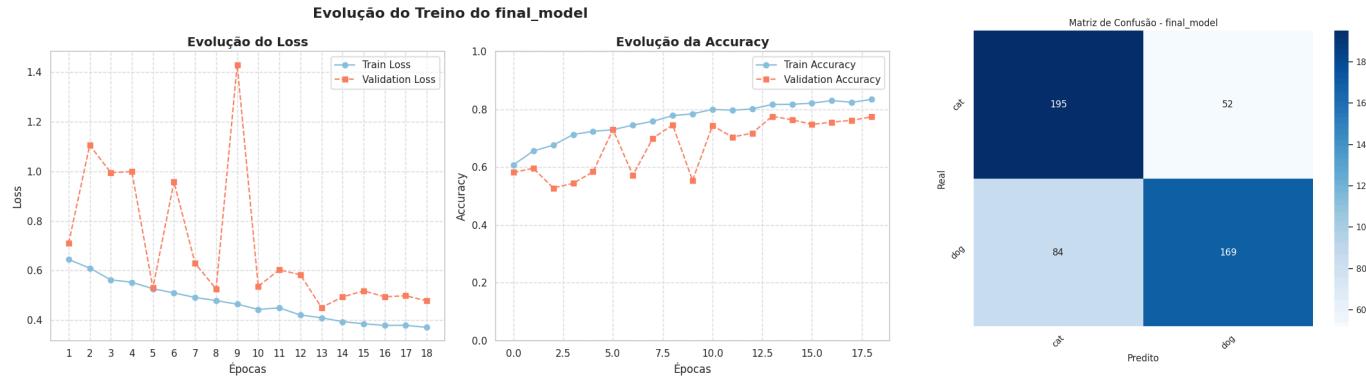


Figura 4: Resultados (Evolução do Treino e Matriz de confusão no teste) para o modelo com *Batch Normalization* (5)

Na Figura 4 verificamos picos nas curvas de validação para ambas as métricas, ao contrário da curva de treino que decresce ou cresce de forma constante, respectivamente. Este comportamento é justificado pelo pequeno número de observações dos dados de validação (1), isto é, as alterações feitas nos pesos do modelo são adequadas para os dados de treino e não para os de validação. No conjunto de teste, vemos que a matriz de confusão faz previsões sólidas, numa taxa de acertos de cerca de 72.8%, valor ligeiramente mais baixo que o mostrado no gráfico da evolução da *accuracy* na melhor época (13-menor valor de *loss*). Esta diferença pode ser explicada pela disparidade das imagens no conjunto de teste, o que torna a generalização mais “desafiante” para o modelo.

$$\text{Conj-Treino} = \frac{2000}{3000} \times 100\% \approx 66.7\% >> \text{ConjValidação} = \frac{500}{3000} \times 100\% \approx 16.7\% \quad (1)$$



Figura 5: Mapa de saliência de classificações incorretas para o modelo final com *Batch Normalization* (5)

³Para esta fase do projeto foi implementada uma solução para que os modelos corressem na GPU, algo que diminuiu drasticamente o tempo de execução dos mesmos. Para mais detalhes da implementação ver [README do projeto](#). GPU utilizada para a execução dos modelos “custom”: [GeForce RTX™ 4060 WINDFORCE OC 8G](#)

⁴CPU utilizada para a execução dos modelos “custom”: [Ryzen™ 5 7600X](#)

Ao analisar a [Figura 5](#), com classificações incorretas e os mapas de saliência, observamos que o modelo está focado em regiões exteriores ao gato na 1^a imagem (talvez pela luminosidade) e, por isso, classifica como cão com 0.738 de probabilidade. Já na 2^a imagem, apesar de encontrar com sucesso o animal, não consegue extrair adequadamente as características do mesmo para prever na classe correta.

Parte 3 : Transferência de conhecimento

Com o objetivo de melhorar a capacidade de classificação do nosso modelo, é neste segmento aplicada a transferência de conhecimento. Esta abordagem aproveita redes pré-treinadas, que possuem um nível superior de complexidade na extração de características. Desta forma, o treino será focado na otimização do classificador.

Estas transferências foram realizadas com as redes [MobileNetV3 Small](#), [MobileNetV3 Large](#), [VGG16](#) e [ResNet50](#).

Tabela 2: Resultados dos modelos pré-treinados com o treino da rede convolucional

Modelo Pré-treinado	Tempo de Execução GPU ⁵	Nº Epochs	Métricas de Desempenho			Nº Parâmetros	
			Precision	Accuracy	Recall	Treináveis	Totais
MobileNetV3 Small	20.9s	8	0.984	0.984	0.984	4 718 849	15 095 669
MobileNetV3 Large	1m 29.9s	13	1	0.996	0.992	7 864 577	26 590 085
VGG16	1m 13.8s	10	0.984	0.968	0.953	3 211 521	24 349 253
ResNet50	1m 19.9s	13	0.98	0.982	0.984	12 845 313	62 123 653

Com base na [Tabela 2](#) conseguimos perceber que o MobileNetV3 Large destaca-se com 99.6% de Accuracy e 100% de Precision, seguido pelo ResNet50 e VGG16, que também apresentam métricas acima de 96%. Ainda, é possível comparar a versão do modelo MobileNetV3 Large com o Small, onde verificamos que o Large destaca-se por ter um desempenho ligeiramente superior, mas com um custo computacional mais elevado ($\approx 4.5x$ maior), enquanto o Small pode ser mais útil para cenários com restrições de recursos sem comprometer significativamente as métricas. Em geral, os modelos revelam todos muito boas previsões e erram muito pouco (de 2-16 observações → [Anexo C](#)).

É de se notar também que houve uma melhoria bastante significativa deste modelos face aos resultados da rede convolucional “custom” desenvolvida na [Parte 2](#), especialmente nas métricas de desempenho. Talvez esta discrepancia de valores deva-se ao facto de que estes modelos pré-treinados já foram treinados em grandes bases de dados, o que permite uma extração mais aprofundada de características e generalização. Além disso, os tempos de execução são bastante satisfatórios, tornando estes modelos em opções viáveis para aplicações práticas como esta.

A imagem mostra a saída de um modelo pré-treinado MobileNetV3Large, que classificou corretamente a imagem como um gato. A saliência indica que o modelo focou principalmente na região do olho e um pouco no formato da orelha do gato, o que já foi suficiente para garantir uma classificação precisa, com 100% de probabilidade. Estas características fisiológicas são essenciais para diferenciar um gato de um cão, o que reforça a eficiência deste modelo em classificação de imagens.

Modelo utilizado - MobileNetV3Large

Original Image: cat



Probabilidade dog: 0.000



Figura 6: Exemplo de imagem que o MobileNetV3 Large prevê corretamente com 100% de probabilidade

⁵GPU utilizada para execução dos modelos pré-treinados: [GeForce RTX™ 4060 WINDFORCE OC 8G](#)

Parte 4 : Resultados num conjunto out-of-sample

Nesta fase, serão utilizadas imagens de cães e gatos capturadas pelo grupo para realizar previsões e analisar o desempenho dos modelos numa amostra totalmente diferente. Para isso, será dado um maior foco nos dois modelos mais robustos de cada abordagem: o modelo “custom” Final Model, desenvolvido na [Parte 2](#) (com métricas apresentadas em [Tabela 1](#)), e o modelo de “transfer learning” MobileNetV3 Large, criado na [Parte 3](#) (com métricas disponíveis em [Tabela 2](#)). As imagens utilizadas foram previamente cortadas (*cropped*) para evitar o achatamento na leitura e *rescale* das imagens e foram no total **16 imagens**, 7 imagens de cães e 9 de gatos.

“Custom” Model - Final Model (5) :

Errou 2 gatos (acertando 7) e 3 cães (acertando 4). Ou seja, esta previsão registou 0.778 de *precision*, 0.625 de *accuracy* e 0.636 de *recall*. Na [Figura 7](#) observamos dois exemplos de imagens más classificadas que captam características diferentes. A da Luna (imagem da esquerda) aparenta focar-se em demasia na “portinhola” da gaiola e não no animal. Já a imagem à direita foca-se no animal, contudo, fatores como a luminosidade ou a pequena dimensão do mesmo, podem resultar na previsão errada.

Modelo utilizado - models/final_model.keras



Modelo utilizado - models/final_model.keras



Figura 7: Fotografias previstas de forma incorreta pelo modelo Final model

Transfer Learning Model: MobileNetV3 Large:

Errou 4 gatos (acertando 5) e 2 cães (acertando 5). Ou seja, esta previsão registou 0.556 de *precision*, 0.438 de *accuracy* e 0.5 de *recall*. Na [Figura 8](#), o modelo pré-treinado demonstra uma alta confiança na previsão, assim como na [Figura 6](#), porém, desta vez, erra na classe original. Foi de facto surpreendente o modelo prever incorretamente, pois a saliência parece estar bastante evidente nos pontos característicos que diferem um gato e um cão. Acreditamos que seja pela forma distinta de como estas fotografias estejam representadas em relação às imagens do conjunto de dados.

Modelo utilizado - models/MobileNetV3Large.keras



Modelo utilizado - models/MobileNetV3Large.keras



Figura 8: Fotografias previstas de forma incorreta pelo modelo MobileNetV3 Large

Embora sejam poucas imagens, temos um conjunto de fotografias de fácil e difícil previsão. De modo geral, o Final Model obteve um desempenho superior, possivelmente devido a uma estrutura mais ajustada ao problema. Já o MobileNetV3 Large, por ser um modelo mais pesado, pode ter sido impactado pelo excesso de parametrização e *layers*, bem como pelas condições diferentes na leitura das imagens, o que pode ter impactado negativamente os resultados.

O repositório do projeto pode ser encontrado no seguinte [link](#).

Anexos

Anexo A - Evolução do treino e matriz de confusão nas várias fases do modelo “custom”

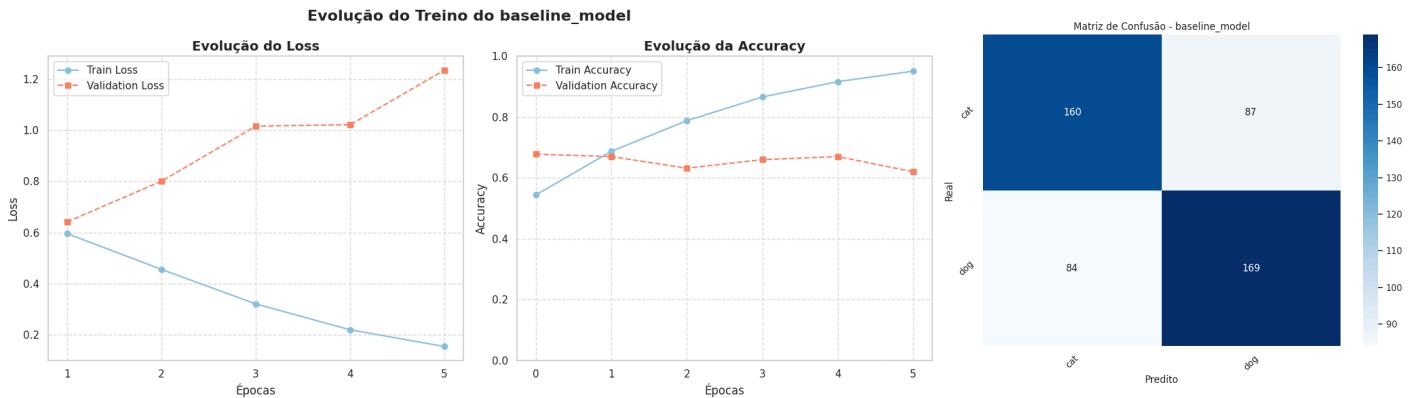


Figura 9: Resultados para o modelo *baseline* (1)

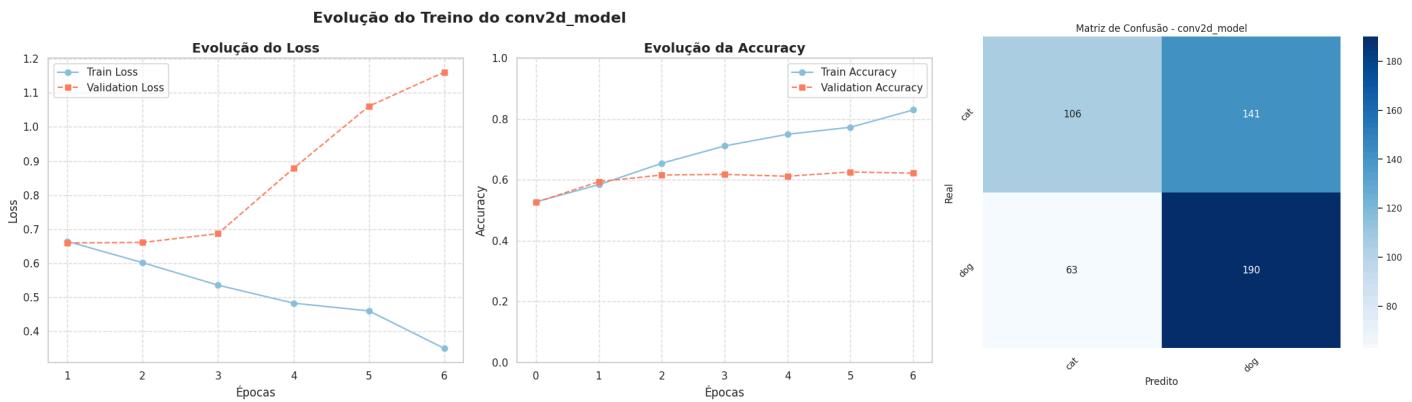


Figura 10: Resultados para o modelo com mais uma camada convolucional (2)

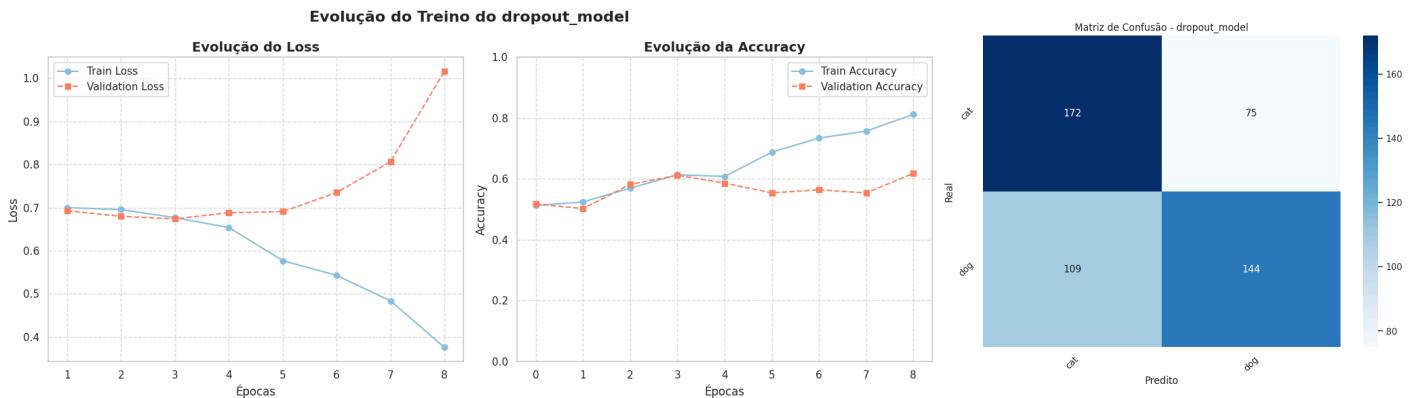


Figura 11: Resultados para o modelo com uma *layer* de *dropout* (3)

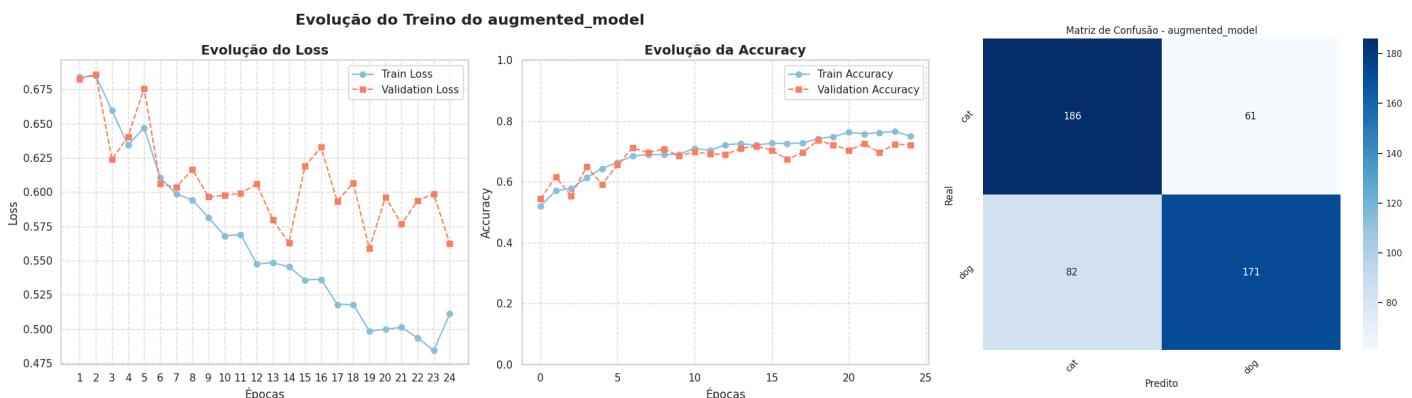


Figura 12: Resultados para o modelo com Data Augmentation (4)

Anexo B - Evolução do treino e matriz de confusão nos modelos pré-treinados

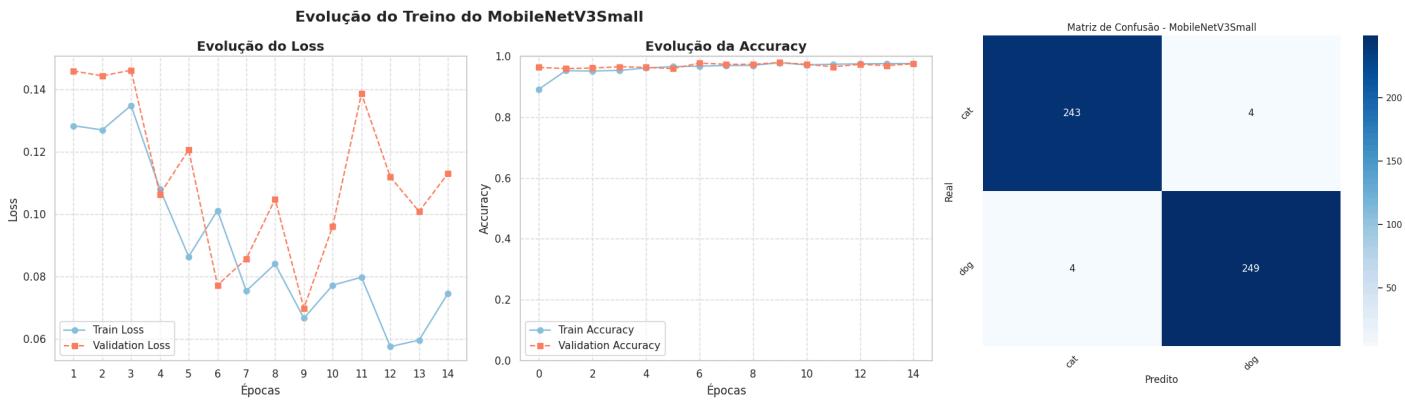


Figura 13: Resultados para o modelo MobileNetV3 Small

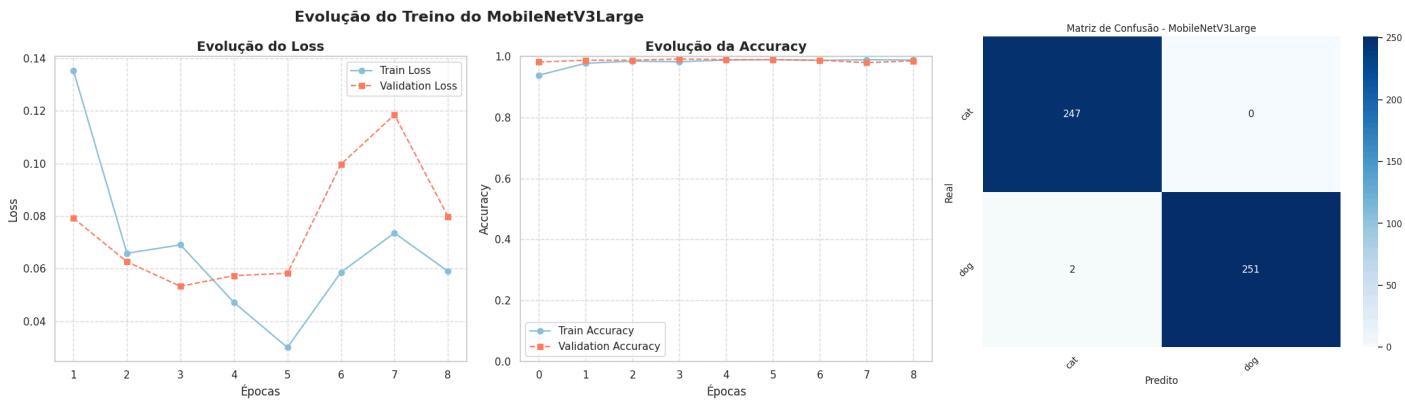


Figura 14: Resultados para o modelo MobileNetV3 Large

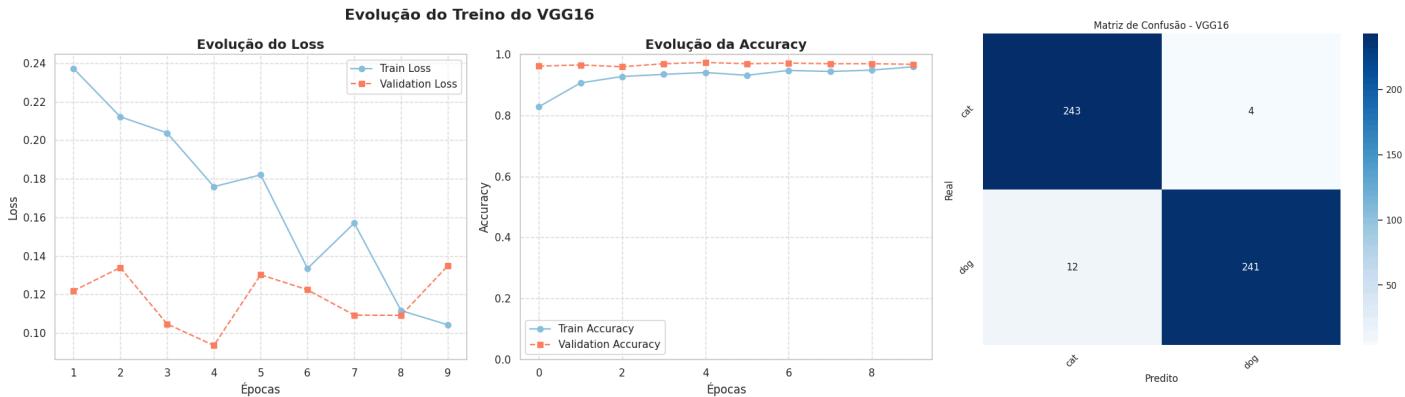


Figura 15: Resultados para o modelo VGG16

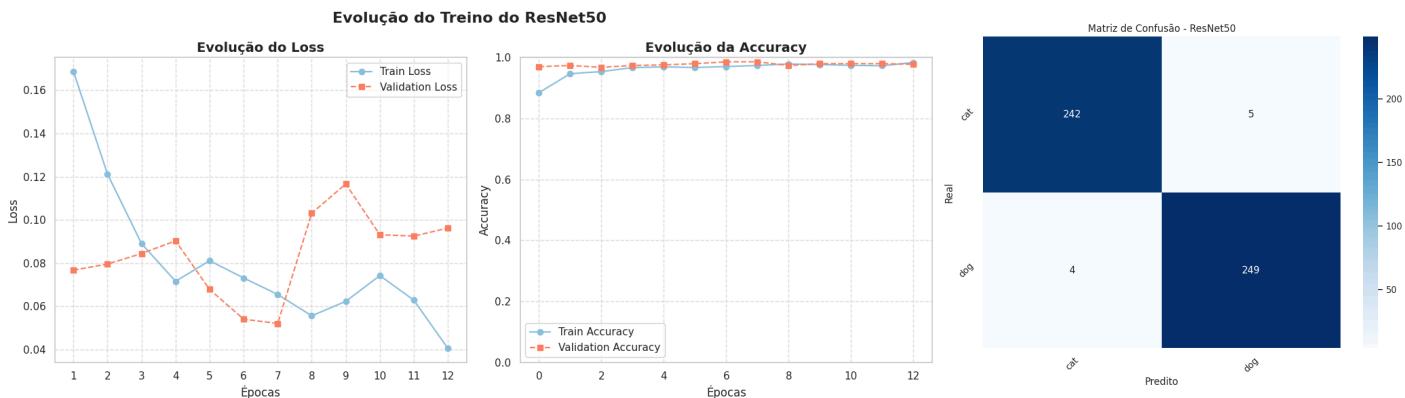


Figura 16: Resultados para o modelo ResNet50

Anexo C - Todas as classificações incorretas nos modelos pré-treinados

Modelo utilizado - MobileNetV3Small

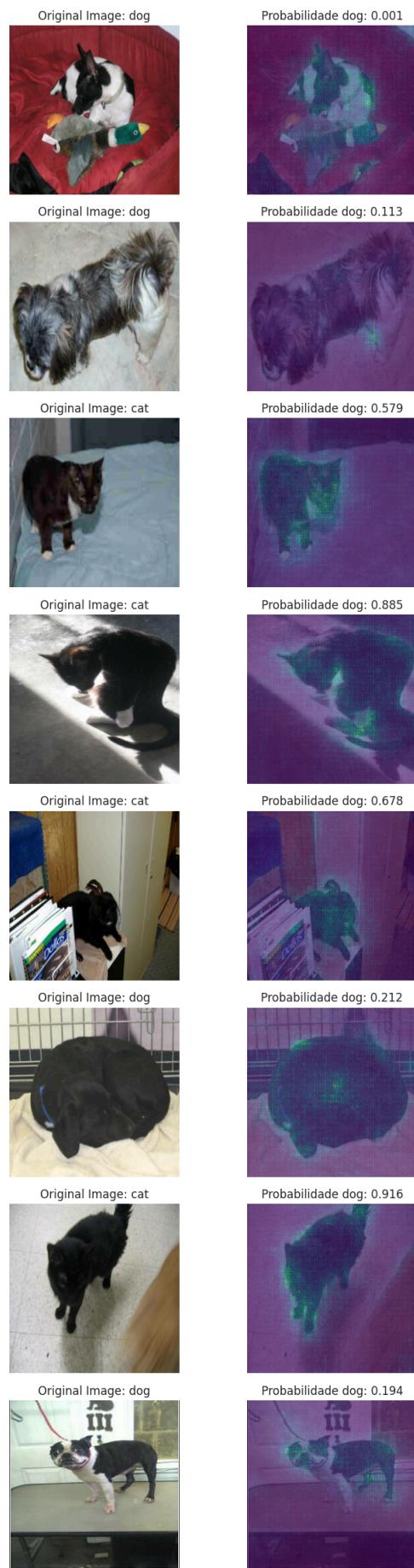


Figura 17: Classificações incorretas e mapa de saliência para o modelo **MobileNetV3 Small**

Modelo utilizado - MobileNetV3Large

Original Image: dog



Probabilidade dog: 0.290



Original Image: dog



Probabilidade dog: 0.009



Figura 18: Classificações incorretas e mapa de saliência para o modelo **MobileNetV3 Large**

Modelo utilizado - VGG16

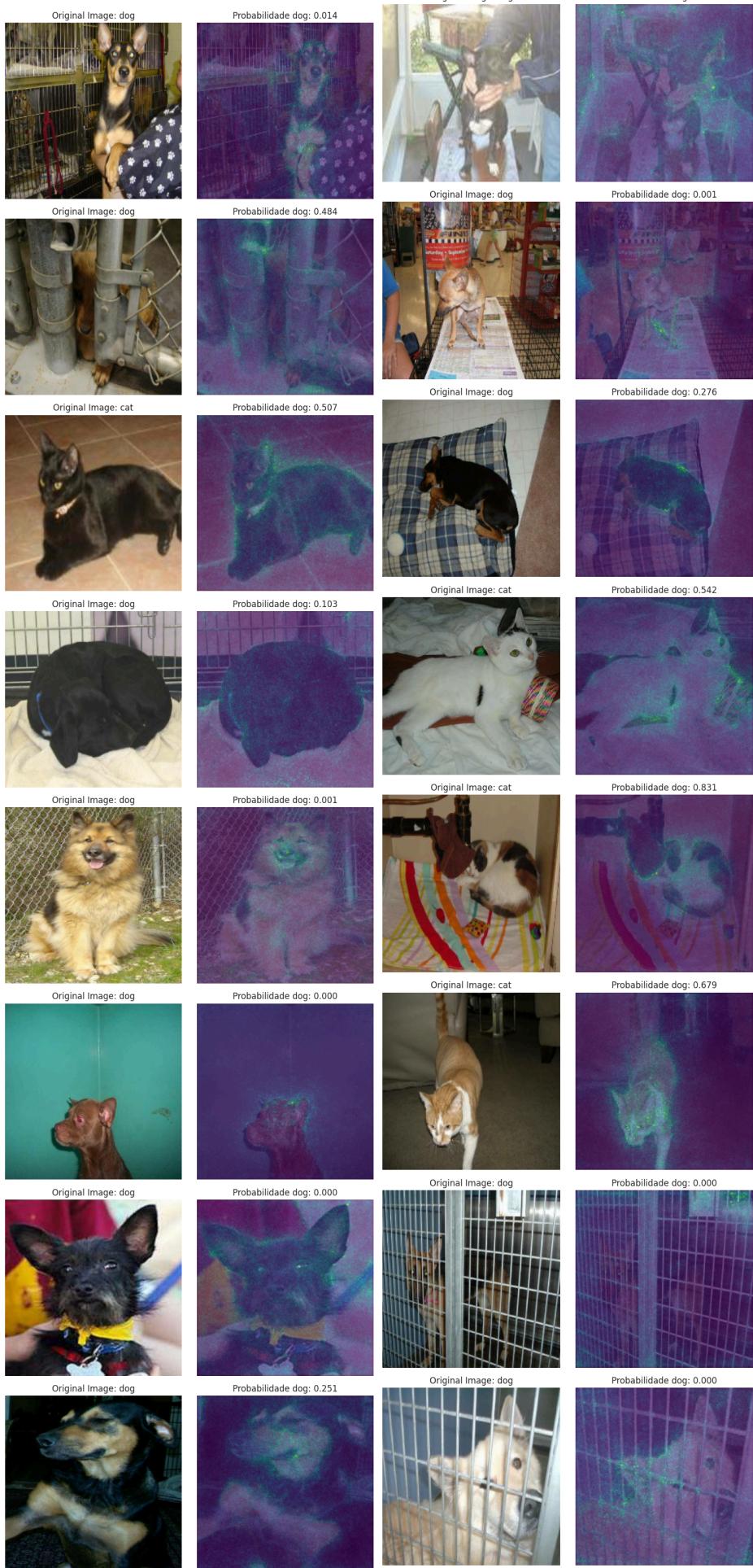


Figura 19: Classificações incorretas e mapa de saliência para o modelo **VGG16**

Modelo utilizado - ResNet50



Figura 20: Classificações incorretas e mapa de saliência para o modelo **ResNet50**

Anexo D - Matrizes de confusão para as imagens out_of_sample

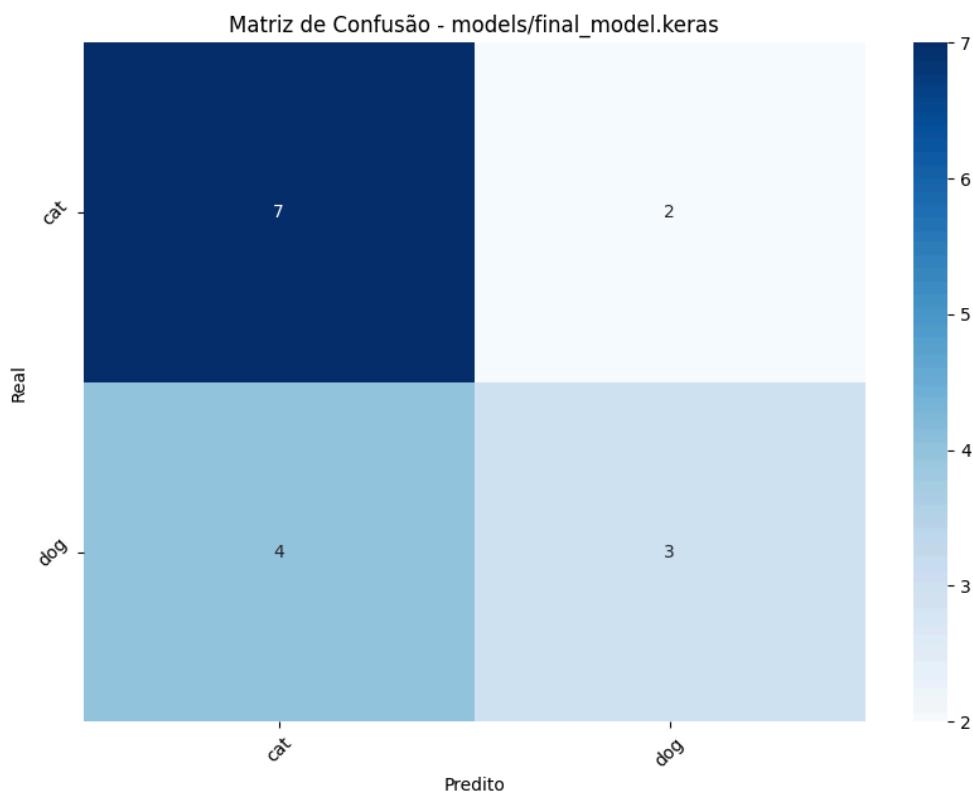


Figura 21: Matriz de confusão para o Final Model

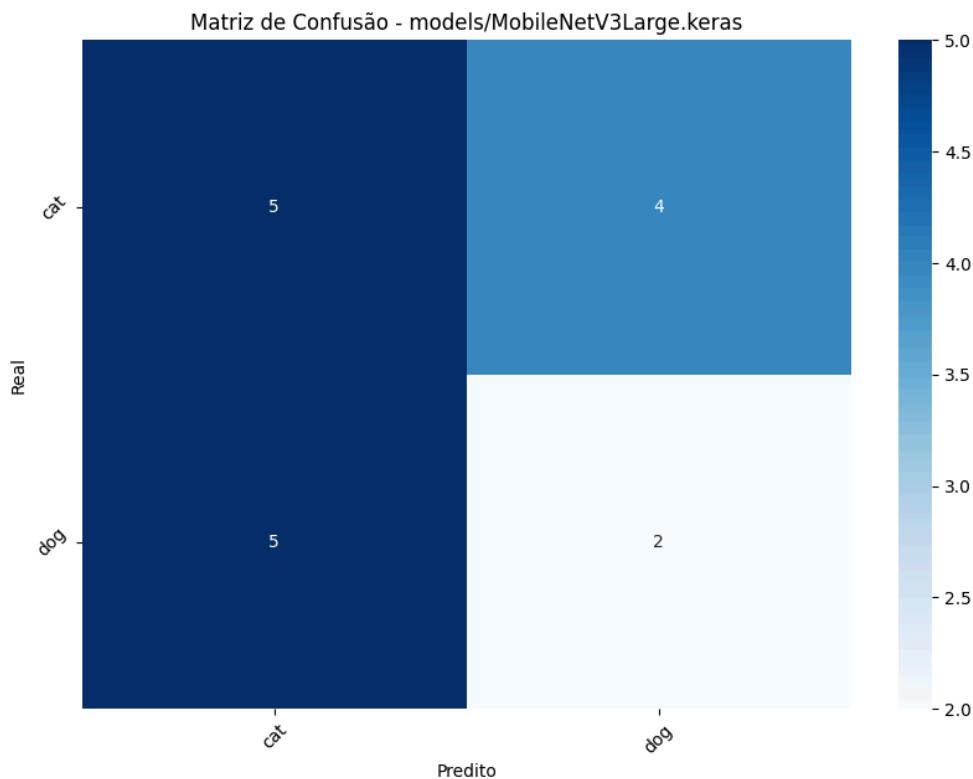


Figura 22: Matriz de confusão para o MobileNetV3 Large

Anexo E - Tentativa de ajuste dos hiperparâmetros do modelo com Optuna

```
1 def create_model(trial: optuna.Trial) -> keras.Sequential:
2     params = {
3         "use_data_augmentation" : trial.suggest_categorical("use_data_augmentation", [False, True]),      # Data Augmentation control
4         "use_dropout"          : trial.suggest_categorical("use_dropout", [False, True]),                  # Dropout control
5         "use_batch_normalization": trial.suggest_categorical("use_batch_normalization", [False, True]),    # Batch Normalization control
6         "n_layers_CNN"         : trial.suggest_int("n_layers_CNN", 1, 6),                                # Number of CNN layers control
7         "pool_every_n_layers"   : trial.suggest_int("pool_every_n_layers", 1, 3),                      # When to pool control
8         "n_layers_hidden"       : trial.suggest_int("n_layers_hidden", 1, 2)                           # Number of dense layers control
9     }
10    model = keras.Sequential([
11        layers.Input(shape=(img_height, img_width, 3)),
12    ])
13    if params["use_data_augmentation"]: # If True, add data augmentation layer
14        model.add(data_augmentation)
15    for i_layer in range(params["n_layers_CNN"]): # For suggested number of CNN layers
16        params[f"CNN_layer{i_layer}"] = {
17            "n_filters"   : trial.suggest_int(f"CNN_layer{i_layer}_n_filters", 4, 64, log=True),           # N filters for current CNN layer
18            "kernel_size" : trial.suggest_int(f"CNN_layer{i_layer}_kernel_size", 2, 6),                   # Kernel size for current layer
19            "activation"  : trial.suggest_categorical(f"CNN_layer{i_layer}_activation", [None, "relu"]) # Activation for current layer
20        }
21    model.add(
22        layers.Conv2D(
23            filters=params[f"CNN_layer{i_layer}"]["n_filters"],
24            kernel_size=params[f"CNN_layer{i_layer}"]["kernel_size"],
25            padding="same",
26            activation=None
27        )
28    )
29    if params["use_batch_normalization"]: # If True, add Batch normalization
30        model.add(layers.BatchNormalization())
31    if params[f"CNN_layer{i_layer}"]["activation"] != None: # If not None, add selected activation layer
32        model.add(layers.Activation(params[f"CNN_layer{i_layer}"]["activation"]))
33    if i_layer % params["pool_every_n_layers"] == 0: # If current layer number is divisible by pooling control, add pooling layer
34        params[f"CNN_layer{i_layer}"]["pool_size"] = trial.suggest_int(f"CNN_layer{i_layer}_pool_size", 1, 3) # Pool size for current layer
35        model.add(layers.MaxPooling2D(params[f"CNN_layer{i_layer}"]["pool_size"]))
36    model.add(layers.Flatten())
37    for i_layer in range(params["n_layers_hidden"]): # For suggested number of hidden layers
38        params[f"hidden_layer{i_layer}"] = {
39            "n_neurons": trial.suggest_int("n_neurons", 2, 256, log=True) # N neurons for current hidden layer
40        }
41        model.add(layers.Dense(params[f"hidden_layer{i_layer}"]["n_neurons"], activation=None))
42        if params["use_batch_normalization"]:
43            model.add(layers.BatchNormalization())
44        model.add(layers.Activation("relu"))
45    if params["use_dropout"]:
46        params["dropout_rate"] = trial.suggest_float("dropout_rate", 0.1, 0.5) # Dropout rate for current model
47        model.add(layers.Dropout(params["dropout_rate"]))
48    model.add(layers.Dense(1, activation="sigmoid")) # Output layer
49    trial.set_user_attr("model_params", params)
50
51    return model
```

Figura 23: Código de *tuning* dos modelos desenvolvidos