SEVEN
publicações acadêmicas

# Aplicações de Redes Neurais Convolucionais no reconhecimento de danos em veículos

Crossref doi
https://doi.org/10.56238/sevened2023.006-146

**Erick Roberto Furst Brito**

**RESUMO**
As Redes Neurais Convolucionais (CNNs) têm demonstrado um desempenho excepcional em uma variedade de tarefas de visão computacional. Este artigo explora o uso de CNNs no contexto do reconhecimento de danos em veículos, com foco na integração de um produto chamado IVI (Inspeção Veicular Inteligente) desenvolvido para essa finalidade.

**Palavras-chave:** CNN, Redes Neurais Convolucionais, Visão computacional, Inspeção veicular.

## 1 INTRODUÇÃO

A inspeção veicular em uma locadora de veículos, por exemplo, é um processo importante que visa garantir a segurança, qualidade e conformidade de cada veículo alugado antes e depois de serem entregues aos clientes. O processo de inspeção pode variar de uma locadora para outra, mas geralmente inclui os seguintes passos:

Check-in do Veículo; Registro de Danos; Registro de Quilometragem; Verificação de Nível de Combustível; Verificação de Documentação e Limpeza e Manutenção Básica.

O processo de inspeção veicular em locadoras é fundamental para garantir a segurança dos clientes, proteger os ativos da locadora e manter a satisfação do cliente. Qualquer dano adicional causado pelo cliente durante o período de aluguel também é registrado e pode resultar em custos adicionais.

Para a etapa de verificação de danos, por vezes podem haver discordâncias quanto à responsabilidade, muitas locadoras atualmente já automatizam a maior parte do processo, deixando o cliente quase em uma situação de self-service, não fosse pela verificação dos danos. A visão computacional através das Redes Neurais Convolucionais, pode ser a resposta para isso.

As redes neurais convolucionais são um subconjunto especializado de redes neurais profundas que têm se destacado na análise de imagens devido à sua capacidade de aprender representações hierárquicas. Em particular, elas têm sido amplamente aplicadas no reconhecimento de objetos em imagens, tornando-se uma escolha natural para o desenvolvimento de soluções de inspeção de veículos.

As redes neurais convolucionais (CNNs, do inglês Convolutional Neural Networks) são um tipo de arquitetura de rede neural projetada especificamente para tarefas de visão computacional, como reconhecimento de padrões em imagens e vídeos. Elas são chamadas de "convolucionais" devido à operação fundamental chamada convolução, que desempenha um papel central nesse tipo de rede.

As CNNs recebem uma imagem como entrada. Cada imagem é representada como uma matriz de pixels, onde cada pixel possui valores que representam sua intensidade de cor (por exemplo, valores de pixel em escala de cinza ou valores de canal RGB para imagens coloridas).

A operação de convolução é aplicada na camada inicial da rede. A convolução envolve uma pequena matriz chamada filtro ou kernel, que desliza pela imagem de entrada para extrair recursos locais. À medida que o filtro se move pela imagem, ele realiza operações de multiplicação e soma para criar um mapa de características.

Os mapas de características resultantes capturam informações relevantes da imagem, como bordas, texturas e padrões simples. Após as camadas convolucionais, é comum usar camadas de pooling para reduzir a dimensionalidade dos mapas de características e extrair informações mais importantes. A camada de pooling (geralmente max-pooling) reduz a resolução espacial, mantendo as características mais importantes.

Depois de várias camadas convolucionais e de pooling, as CNNs geralmente têm camadas totalmente conectadas, semelhantes às redes neurais tradicionais. Essas camadas aprendem a combinar as informações extraídas anteriormente para realizar a classificação final ou outra tarefa desejada.

A última camada da CNN é a camada de saída, que depende da tarefa específica que a rede está resolvendo. Por exemplo, em um problema de classificação de imagem, a camada de saída pode ter um neurônio para cada classe possível, com funções de ativação como softmax para produzir probabilidades de cada classe.

Durante o treinamento, a CNN ajusta seus parâmetros (pesos dos filtros, pesos das conexões nas camadas totalmente conectadas) usando algoritmos de otimização, como gradiente descendente, para minimizar uma função de perda que avalia o quão bem a rede está fazendo sua tarefa.

Realizado o treinamento, a CNN pode ser usada para classificar novas imagens ou realizar outras tarefas relacionadas à visão computacional. A imagem de entrada passa pela rede, e a camada de saída fornece as previsões ou resultados desejados.

As CNNs são altamente eficazes em tarefas de visão computacional devido à sua capacidade de aprender automaticamente características relevantes em diferentes níveis de abstração. Elas revolucionaram campos como reconhecimento de imagem, detecção de objetos, segmentação de imagem e muito mais. Além disso, arquiteturas avançadas, como redes residuais (ResNets) e redes neurais convolucionais profundas (CNNs profundas), tornaram possível o treinamento de redes mais profundas e poderosas para tarefas complexas.

As Redes Neurais Convolucionais, se baseiam em duas operações principais: convolução e agrupamento.

A convolução é uma operação matemática usada para processar uma imagem. Imagine que você tem uma imagem de entrada (uma matriz de pixels) e um conjunto de filtros (kernels) que são matrizes menores. A convolução consiste em deslizar esses filtros sobre a imagem de entrada e realizar multiplicação e soma para obter um mapa de características. Isso pode ser expresso matematicamente da seguinte forma para um único filtro:

$$(I * K)(x, y) = \sum_{i=1}^{H} \sum_{j=1}^{W} I(x + i, y + j) * K(i, j)$$

Onde:

- I é a imagem de entrada.
- K é o filtro (kernel).
- H e W são as dimensões do filtro.

Feita a convolução, é comum aplicar uma operação de agrupamento (Pooling), geralmente o agrupamento máximo (max-pooling). Essa operação reduz o tamanho da saída ao selecionar o valor máximo em uma região específica. Por exemplo, se você usar uma janela 2x2, obterá o valor máximo de cada grupo de 4 valores na saída da convolução.

$$Max - Pooling\ (x, y) = Max(I(x, y), I(x + 1, y), I\ (x, Y + 1), I(x + 1, y + 1))$$

Essas operações são repetidas em camadas sucessivas da CNN para extrair características progressivamente mais abstratas da imagem. Em seguida, essas características são alimentadas em uma rede neural totalmente conectada para a classificação final.

É importante mencionar que em uma CNN completa, vários filtros são usados em cada camada convolucional para capturar diferentes características da imagem, e as operações de convolução e agrupamento são realizadas em paralelo em múltiplos canais de cores se a imagem for colorida (RGB).

## 2 METODOLOGIA

### 2.1 ARQUITETURA DA CNN

Foi criado um modelo na linguagem de programação Python 3.5, se utilizando da biblioteca Keras. O peso utilizado foi baixado de uma página do Git Hub, com modelos pré-treinados (https://github.com/fchollet/deep-learning-models/releases/) para Keras, Tensorflow. A VGG16 é uma arquitetura de rede neural convolucional popular para tarefas de visão computacional. É conhecida por

sua simplicidade relativa e bom desempenho em tarefas de classificação de imagens. O modelo inicial simplesmente identifica um veículo danificado de um não danificado, e posteriormente evoluiu para um modelo identificando os tipo de danos apresentados, para efeitos de simplicidade e cronologia, irei apresentar os principais trechos do código para diferenciação de veículo danificado para não danificado.

O modelo inicialmente criado usando a arquitetura VGG16 pré-treinada:

Código:

```
base_model = VGG16(weights='imagenet',include_top=False,input_shape=input_shape
,pooling=max)
```

Foram utilizados pesos pré-treinados da VGG16 que foram treinados na tarefa de classificação de imagem no conjunto de dados ImageNet. Usar pesos pré-treinados é uma prática comum ao treinar modelos de visão computacional, pois esses pesos contêm informações úteis aprendidas com um grande conjunto de dados.

Também se optou por não incluir a camada densa (totalmente conectada) no topo da rede neural. Em vez disso, utilizei a arquitetura da VGG16 até a última camada convolucional. Isso é útil quando pretendemos personalizar a rede para uma tarefa específica, como transferência de aprendizado (Transfer Learning).

A camada de pooling a ser utilizada após as camadas convolucionais, foi configurada para "max", o que significa que a camada de pooling usará a operação de máximo para reduzir as dimensões espaciais dos recursos convolucionais.

Em resumo, o código cria um modelo base VGG16 com pesos pré-treinados da ImageNet, remove a camada densa superior, define o formato das imagens de entrada e especifica o uso da camada de pooling máxima após as camadas convolucionais. Este modelo pode ser usado como base para tarefas de transferência de aprendizado, onde você pode adicionar suas próprias camadas personalizadas no topo para treiná-lo em uma tarefa específica, como classificação de imagens.

Código:

```
for layer in base_model.layers:
layer.trainable=False
```

Desativação do treinamento das camadas do modelo base, para que apenas as camadas adicionadas posteriormente (camadas personalizadas) sejam treinadas durante o processo de treinamento global da rede. Isso é particularmente útil quando você tem um conjunto de dados menor

ou uma tarefa de classificação de imagens relacionada àquelas que o modelo base já aprendeu. Congelar as camadas do modelo base ajuda a preservar os recursos e o conhecimento que ele já adquiriu, ao mesmo tempo em que permite que você adapte o modelo para sua tarefa específica.

A razão para fazer isso é utilizar o método de "Transfer Learning". A ideia por trás da transferência de aprendizado é que você pode usar um modelo pré-treinado, como o VGG16 com pesos da ImageNet, que já foram treinados com uma base de dados sólida que já aprendeu a extrair características úteis de imagens. Em vez de treinar todo o modelo do zero (o que pode ser caro computacionalmente), você pode congelar as camadas do modelo base para que elas não sejam alteradas e, em seguida, adicionar camadas personalizadas no topo para aprender a tarefa específica que você deseja.

Código:

**base_model.summary()**

O modelo base gerado:

Model: "vgg16"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_3 (InputLayer) | (None, 256, 256, 3) | 0 |
| block1_conv1 (Conv2D) | (None, 256, 256, 64) | 1792 |
| block1_conv2 (Conv2D) | (None, 256, 256, 64) | 36928 |
| block1_pool (MaxPooling2D) | (None, 128, 128, 64) | 0 |
| block2_conv1 (Conv2D) | (None, 128, 128, 128) | 73856 |
| block2_conv2 (Conv2D) | (None, 128, 128, 128) | 147584 |
| block2_pool (MaxPooling2D) | (None, 64, 64, 128) | 0 |
| block3_conv1 (Conv2D) | (None, 64, 64, 256) | 295168 |
| block3_conv2 (Conv2D) | (None, 64, 64, 256) | 590080 |
| block3_conv3 (Conv2D) | (None, 64, 64, 256) | 590080 |
| block3_pool (MaxPooling2D) | (None, 32, 32, 256) | 0 |
| block4_conv1 (Conv2D) | (None, 32, 32, 512) | 1180160 |
| block4_conv2 (Conv2D) | (None, 32, 32, 512) | 2359808 |
| block4_conv3 (Conv2D) | (None, 32, 32, 512) | 2359808 |
| block4_pool (MaxPooling2D) | (None, 16, 16, 512) | 0 |

block5_conv1 (Conv2D) (None, 16, 16, 512) 2359808

block5_conv2 (Conv2D) (None, 16, 16, 512) 2359808

block5_conv3 (Conv2D) (None, 16, 16, 512) 2359808

block5_pool (MaxPooling2D) (None, 8, 8, 512) 0

==========

Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688

## 3 TREINAMENTO

E etapa consistiu em treinar um modelo de classificação binária (ou seja, um modelo que classifica amostras em duas classes (Danificado ou não danificado). Inicialmente carrega-se as características de bottleneck (representações intermediárias) das imagens de treinamento que foram calculadas e salvas anteriormente em arquivos .npy. Essas características servirão como entrada para o modelo de classificação.

O array numpy “train_labels” contém rótulos de classe para as amostras de treinamento. Esses rótulos são criados com base no número de amostras positivas e negativas, onde 0 representa "Dano" e 1 "Sem Dano". O array numpy “val_labels” contém rótulos de classe para as amostras de validação, seguindo um padrão semelhante ao das amostras de treinamento.

Foi criado um modelo de rede neural sequencial (Sequential) com uma camada de entrada (Flatten), uma camada densa (Dense) com ativação ReLU, uma camada de dropout (Dropout) e uma camada densa de saída com ativação sigmoid.

A compilação configurou o modelo para o treinamento, especificando o otimizador, a função de perda/loss (binary_crossentropy para classificação binária) e as métricas a serem monitoradas (precisão).

O checkpoint definiu um callback que serviu para monitorar a métrica de precisão na validação (val_acc) e salvar os pesos do modelo sempre que a precisão na validação melhorar. Isso ajuda a evitar o sobreajuste (overfitting) e permite o acompanhamento do modelo durante o treinamento. O treinamento ocorre pelo número de épocas (Epochs) especificado por nb_epoch.

O parâmetro "Lr" refere-se à taxa de aprendizado (learning rate) do otimizador SGD (Stochastic Gradient Descent) usado para treinar a rede neural. A taxa de aprendizado é um hiperparâmetro crucial no treinamento de redes neurais e controla o tamanho dos passos que o otimizador dá ao ajustar os pesos da rede durante o treinamento. Uma taxa de aprendizado menor, como 0.0001, significa que os pesos da rede são ajustados com pequenos passos a cada iteração de treinamento. Isso pode ajudar a convergência do modelo, especialmente quando se treina em conjuntos de dados complexos. No entanto, a escolha da taxa de aprendizado adequada pode ser um desafio e muitas vezes requerem ajustes empíricos para obter o melhor desempenho do modelo.

É importante notar que a taxa de aprendizado é apenas um dos hiperparâmetros que afetam o treinamento de redes neurais, e encontrar a combinação certa de hiperparâmetros para um problema específico é uma parte crítica do desenvolvimento de modelos de aprendizado de máquina eficazes.

Finalmente, a função retorna o modelo treinado e o histórico de treinamento (fit.history), que contém informações sobre as métricas de treinamento ao longo das épocas.

Código:

```
def train_binary_model(location):
train_data = np.load(open(location+'/bottleneck_features_train.npy', 'rb'))
print(train_data.shape[1:])
train_labels = np.array([0]*train_samples[0]+
[1]*train_samples[1])

#Deleção para equiparar o tamanho dos arrays
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])
train_labels = np.delete(train_labels,[1])

val_data = np.load(open(location+'/bottleneck_features_val.npy', 'rb'))
val_labels = np.array([0]*val_samples[0] +[1]*val_samples[1])

model = Sequential()
model.add(Flatten(input_shape=(train_data.shape[1:])))
model.add(Dense(units=256, activation='relu',
kernel_regularizer=l2(l=0.01)))
model.add(Dropout(rate=0.5))
model.add(Dense(units=1, activation='sigmoid'))

model.compile(optimizer=optimizers.SGD(lr=0.0001, momentum=0.9),
loss='binary_crossentropy',
metrics=['accuracy'])
```

checkpoint = ModelCheckpoint(top_model_weights_path,
monitor='val_acc',
verbose=1,
save_best_only=True,
save_weights_only=True,
mode='auto')

fit = model.fit(train_data, train_labels, epochs=nb_epoch, batch_size=16,
validation_data=(val_data, val_labels), callbacks=[checkpoint])

return model, fit.history

## 3.1 EXECUÇÃO DO TREINAMENTO

Treinamento e armazenamento dos resultados para avaliação posterior. A variável "var_modelo" conterá o modelo treinado e a utilizaremos para realizar previsões, e a variável "var_history" para analisar o desempenho do modelo ao longo das épocas (Epochs) de treinamento.

Código:

**var_model,var_history = train_binary_model(location)**

O treinamento:

**(8, 8, 512)**
**Train on 6880 samples, validate on 460 samples**
**Epoch 1/50**
**6880/6880 [==============================] - 99s 14ms/step - loss: 6.2288 - accuracy: 0.7651 - val_loss: 5.2827 - val_accuracy: 0.8978**
**Epoch 2/50**
**16/6880 [..............................] - ETA: 1:08 - loss: 5.4669 - accuracy: 0.8125**
**c:\users\erick\appdata\local\programs\python\python35\lib\site-packages\keras\callbacks\callbacks.py:707: RuntimeWarning: Can save best model only with val_acc available, skipping.**
**'skipping.' % (self.monitor), RuntimeWarning)**

**6880/6880 [==============================] - 82s 12ms/step - loss: 5.3260 - accuracy: 0.8433 - val_loss: 5.1609 - val_accuracy: 0.9000**

**Epoch 3/50**

**6880/6880 [==============================] - 94s 14ms/step - loss: 5.1949 - accuracy: 0.8660 - val_loss: 5.0686 - val_accuracy: 0.9196**

**Epoch 4/50**

**6880/6880 [==============================] - 121s 18ms/step - loss: 5.0727 - accuracy: 0.8850 - val_loss: 4.9881 - val_accuracy: 0.9152**

**Epoch 5/50**

**6880/6880 [==============================] - 102s 15ms/step - loss: 4.9572 - accuracy: 0.9006 - val_loss: 4.9027 - val_accuracy: 0.9196**

**Epoch 6/50**

**6880/6880 [==============================] - 90s 13ms/step - loss: 4.8471 - accuracy: 0.9153 - val_loss: 4.8216 - val_accuracy: 0.9304**

**Epoch 7/50**

**6880/6880 [==============================] - 89s 13ms/step - loss: 4.7456 - accuracy: 0.9267 - val_loss: 4.7501 - val_accuracy: 0.9239**

**Epoch 8/50**

**6880/6880 [==============================] - 90s 13ms/step - loss: 4.6551 - accuracy: 0.9362 - val_loss: 4.6851 - val_accuracy: 0.9196**

**Epoch 9/50**

**6880/6880 [==============================] - 90s 13ms/step - loss: 4.5573 - accuracy: 0.9451 - val_loss: 4.6128 - val_accuracy: 0.9174**

**Epoch 10/50**

**6880/6880 [==============================] - 89s 13ms/step - loss: 4.4670 - accuracy: 0.9526 - val_loss: 4.5313 - val_accuracy: 0.9326**

**Epoch 11/50**

**6880/6880 [==============================] - 84s 12ms/step - loss: 4.3788 - accuracy: 0.9583 - val_loss: 4.4689 - val_accuracy: 0.9217**

**Epoch 12/50**

**6880/6880 [==============================] - 82s 12ms/step - loss: 4.2981 - accuracy: 0.9634 - val_loss: 4.4295 - val_accuracy: 0.9196 -**

**Epoch 13/50**

**6880/6880 [==============================] - 80s 12ms/step - loss: 4.2242 - accuracy: 0.9638 - val_loss: 4.3494 - val_accuracy: 0.9283**

**Epoch 14/50**

**6880/6880 [==============================] - 80s 12ms/step - loss: 4.1421 - accuracy: 0.9693 - val_loss: 4.2752 - val_accuracy: 0.9261**

**Epoch 15/50**

**6880/6880 [==============================] - 83s 12ms/step - loss: 4.0781 - accuracy: 0.9673 - val_loss: 4.2092 - val_accuracy: 0.9196**

**Epoch 16/50**

**6880/6880 [==============================] - 80s 12ms/step - loss: 4.0036 - accuracy: 0.9725 - val_loss: 4.1435 - val_accuracy: 0.9196**

**Epoch 17/50**

**6880/6880 [==============================] - 80s 12ms/step - loss: 3.9303 - accuracy: 0.9766 - val_loss: 4.0915 - val_accuracy: 0.9370- accura - ETA: 1s - loss: 3.9303 - ac**

**Epoch 18/50**

**6880/6880 [==============================] - 81s 12ms/step - loss: 3.8652 - accuracy: 0.9744 - val_loss: 4.0312 - val_accuracy: 0.9304**

**Epoch 19/50**

**6880/6880 [==============================] - 89s 13ms/step - loss: 3.7944 - accuracy: 0.9786 - val_loss: 3.9783 - val_accuracy: 0.9196**

**Epoch 20/50**

**6880/6880 [==============================] - 93s 14ms/step - loss: 3.7252 - accuracy: 0.9810 - val_loss: 3.9307 - val_accuracy: 0.9261**

**Epoch 21/50**

**6880/6880 [==============================] - 97s 14ms/step - loss: 3.6584 - accuracy: 0.9839 - val_loss: 3.8585 - val_accuracy: 0.9261**

**Epoch 22/50**

**6880/6880 [==============================] - 99s 14ms/step - loss: 3.5975 - accuracy: 0.9828 - val_loss: 3.8325 - val_accuracy: 0.9196**

**Epoch 23/50**

**6880/6880 [==============================] - 96s 14ms/step - loss: 3.5351 - accuracy: 0.9843 - val_loss: 3.7425 - val_accuracy: 0.9283**

**Epoch 24/50**

**6880/6880 [==============================] - 95s 14ms/step - loss: 3.4807 - accuracy: 0.9836 - val_loss: 3.6975 - val_accuracy: 0.9370**

**Epoch 25/50**

**6880/6880 [==========] - 100s 15ms/step - loss: 3.4118 - accuracy: 0.9885 - val_loss: 3.6554 - val_accuracy: 0.9217**

**Epoch 26/50**

**6880/6880 [==========] - 99s 14ms/step - loss: 3.3635 - accuracy: 0.9830 - val_loss: 3.5809 - val_accuracy: 0.9261**

**Epoch 27/50**

**6880/6880 [==========] - 99s 14ms/step - loss: 3.2972 - accuracy: 0.9887 - val_loss: 3.5741 - val_accuracy: 0.9174**

**Epoch 28/50**

**6880/6880 [==========] - 98s 14ms/step - loss: 3.2450 - accuracy: 0.9869 - val_loss: 3.5177 - val_accuracy: 0.9239**

**Epoch 29/50**

**6880/6880 [==========] - 98s 14ms/step - loss: 3.1919 - accuracy: 0.9884 - val_loss: 3.4644 - val_accuracy: 0.9283**

**Epoch 30/50**

**6880/6880 [==========] - 101s 15ms/step - loss: 3.1422 - accuracy: 0.9831 - val_loss: 3.3523 - val_accuracy: 0.9326**

**Epoch 31/50**

**6880/6880 [==========] - 99s 14ms/step - loss: 3.0851 - accuracy: 0.9872 - val_loss: 3.3435 - val_accuracy: 0.9304**

**Epoch 32/50**

**6880/6880 [==========] - 101s 15ms/step - loss: 3.0365 - accuracy: 0.9868 - val_loss: 3.2655 - val_accuracy: 0.9348**

**Epoch 33/50**

**6880/6880 [==========] - 101s 15ms/step - loss: 2.9798 - accuracy: 0.9872 - val_loss: 3.2510 - val_accuracy: 0.9130**

**Epoch 34/50**

**6880/6880 [==========] - 102s 15ms/step - loss: 2.9298 - accuracy: 0.9887 - val_loss: 3.1860 - val_accuracy: 0.9217**

**Epoch 35/50**

**6880/6880 [==========] - 102s 15ms/step - loss: 2.8774 - accuracy: 0.9906 - val_loss: 3.1575 - val_accuracy: 0.9348**

**Epoch 36/50**

**6880/6880 [==========] - 102s 15ms/step - loss: 2.8266 - accuracy: 0.9913 - val_loss: 3.1231 - val_accuracy: 0.9239**

**Epoch 37/50**

**6880/6880 [==============================] - 102s 15ms/step - loss: 2.7829 - accuracy: 0.9910 - val_loss: 3.0363 - val_accuracy: 0.9217**

**Epoch 38/50**

**6880/6880 [==============================] - 105s 15ms/step - loss: 2.7387 - accuracy: 0.9898 - val_loss: 2.9858 - val_accuracy: 0.9391**

**Epoch 39/50**

**6880/6880 [==============================] - 101s 15ms/step - loss: 2.6905 - accuracy: 0.9916 - val_loss: 2.9569 - val_accuracy: 0.9283**

**Epoch 40/50**

**6880/6880 [==============================] - 101s 15ms/step - loss: 2.6458 - accuracy: 0.9900 - val_loss: 2.9396 - val_accuracy: 0.9174**

**Epoch 41/50**

**6880/6880 [==============================] - 101s 15ms/step - loss: 2.6027 - accuracy: 0.9913 - val_loss: 2.8556 - val_accuracy: 0.9348**

**Epoch 42/50**

**6880/6880 [==============================] - 106s 15ms/step - loss: 2.5620 - accuracy: 0.9890 - val_loss: 2.8339 - val_accuracy: 0.9196**

**Epoch 43/50**

**6880/6880 [==============================] - 102s 15ms/step - loss: 2.5147 - accuracy: 0.9907 - val_loss: 2.8269 - val_accuracy: 0.9239**

**Epoch 44/50**

**6880/6880 [==============================] - 102s 15ms/step - loss: 2.4753 - accuracy: 0.9907 - val_loss: 2.7572 - val_accuracy: 0.9239**

**Epoch 45/50**

**6880/6880 [==============================] - 103s 15ms/step - loss: 2.4311 - accuracy: 0.9910 - val_loss: 2.7110 - val_accuracy: 0.9217**

**Epoch 46/50**

**6880/6880 [==============================] - 100s 15ms/step - loss: 2.3947 - accuracy: 0.9911 - val_loss: 2.6739 - val_accuracy: 0.9283**

**Epoch 47/50**

**6880/6880 [==============================] - 102s 15ms/step - loss: 2.3542 - accuracy: 0.9904 - val_loss: 2.6531 - val_accuracy: 0.9196**

**Epoch 48/50**

**6880/6880 [==========] - 102s 15ms/step - loss: 2.3151 - accuracy: 0.9904 - val_loss: 2.5475 - val_accuracy: 0.9370**

**Epoch 49/50**

**6880/6880 [==========] - 101s 15ms/step - loss: 2.2755 - accuracy: 0.9911 - val_loss: 2.5505 - val_accuracy: 0.9348**

**Epoch 50/50**

**6880/6880 [==========] - 109s 16ms/step - loss: 2.2423 - accuracy: 0.9898 - val_loss: 2.5155 - val_accuracy: 0.9304**

## 4 CONJUNTO DE DADOS

Nesta seção, descreveremos em detalhes o processo de coleta de dados para treinar o modelo de identificação de veículos danificados. O conjunto de dados desempenha um papel fundamental no desenvolvimento de qualquer sistema de aprendizado de máquina, e é crucial documentar como as amostras foram obtidas para garantir a transparência e a reprodutibilidade do experimento.

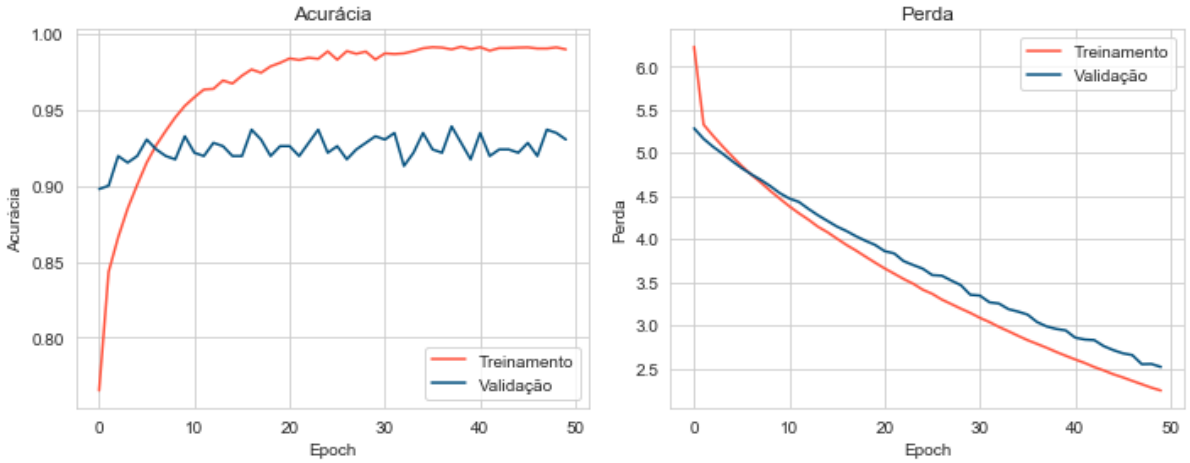
Foram obtidas 4.351 (quatro mil trezentas e cinquenta e uma) imagens de veículos danificados, através de pesquisas no site de buscas Google e de algumas poucas centenas desta amostra através de fotos tiradas de uma aparelho celular de um pátio de uma locadora de veículos e 2.536 (duas mil, quinhentas e trinta e seis) imagens de veículos sem danos obtidas da mesma forma. Essas imagens foram separadas em pastas, de forma manual, por mim. Sendo duas pastas, uma nomeada “Dano” e outra nomeada “SemDano”.

Sendo que a pasta “Dano” continha um tamanho total de 503 mega bytes (527.775.421 bytes) e a pasta “SemDano” um tamanho total de 376 mega bytes (395.072.008 bytes).

## 5 RESULTADOS

O modelo com melhor acurácia 93% (0.939139425453186) foi obtido na 38ª época, com uma perda de 2.98% (2.9858194682909094) - Imagem 1 -. Para a validação foram utilizadas 230 (Duzentas e trinta) imagens. Destas 230 imagens, 6.5% eram somente de arranhados sem qualquer outro tipo de dano, destes nenhum foi classificado de forma errada.

Imagem 1




Fonte: Dados da Pesquisa

Imagem gerada pela aplicação de função de plotagem (plot, import matplotlib.pyplot as plt) com dados a partir da variável “var_history” preenchida durante a fase de treinamento.

O gráfico à esquerda exibe as taxas de acurácia, através das épocas (Epochs) de treinamento.

O gráfico à direita e exibe a taxa de perda (Loss), através das épocas (Epochs) de treinamento.

A tabela verdade (Imagem 2) após o fine tune apresentou os seguintes resultados, com os dados de validação:

Imagem 2


Fonte: Dados da Pesquisa

Imagem gerada pela aplicação da função heatmap (sns.heatmap, import seaborn as sns), a partir do modelo gerado contra dados de validação.

O gráfico de validação gerado apresenta os seguintes resultados, numa amostra de validação de 460 imagens:

Verdadeiros Positivos: 213

Falsos Positivos: 17

Falsos negativos: 15

Verdadeiros Negativos: 215

Sumarizando a tabela confusão na curva ROC:

Verdadeiro positivo = Sensibilidade

Falso positivo = Especificidade

$$Taxa\ Verdadeiro\ Positivo = \frac{Verdadeiro\ Positivo}{Verdadeiro\ Positivo + Falso\ Negativo}$$

Onde a “Taxa Verdadeiro Positivo” informa a proporção de amostras de “Carros Danificados” corretamente classificados.

$$Taxa\ Falso\ Positivo = 1 - Especificidade = \frac{Falso\ Positivo}{Falso\ Positivo + Verdadeiro\ Negativo}$$

Onde a “Taxa de Falso Positivo” informa a proporção de amostras de “Carros Sem Dano” incorretamente classificados.

A curva ROC, ou "Curva Característica de Operação do Receptor", é uma ferramenta gráfica usada para avaliar o desempenho de classificadores binários, como modelos de aprendizado de máquina, especialmente em problemas de classificação binária (duas classes, geralmente positiva e negativa).

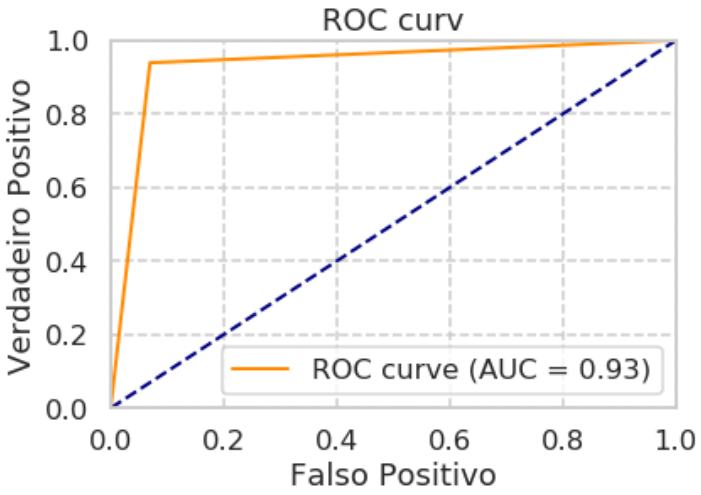
A Curva ROC representa a relação entre a Taxa de Verdadeiros Positivos (True Positive Rate - TPR) e a Taxa de Falsos Positivos (False Positive Rate - FPR) em diferentes pontos de corte (thresholds) para as previsões de um modelo. Uma Curva ROC ideal está mais próxima do canto superior esquerdo do gráfico, onde o TPR é alto e o FPR é baixo. Quanto mais a Curva ROC se desvia dessa linha diagonal, melhor é o desempenho do modelo. O valor da área sob a Curva ROC (AUC-ROC) também é comumente calculado para quantificar o desempenho global do modelo, onde um valor de 1 indica um desempenho perfeito e um valor de 0,5 indica um desempenho aleatório (sem discriminação entre classes).

A linha diagonal azul indica que a Taxa de Verdadeiro Positivo é igual a Taxa de Falsos Positivos, ou seja, o modelo próximo a linha azul é irrelevante, pois apresentaria sempre a mesma proporção de classificações com acertos e erros.

Ou seja, quanto mais afastado a linha diagonal, melhor.

O modelo apresentou um resultado de 0.93 (Imagem 3), um modelo considerado excelente, possuirá a curva ROC o mais próximo de 1 (um).

Imagem 3




Fonte: Dados da Pesquisa

Imagem gerada pela aplicação de função de plotagem (plot, import matplotlib.pyplot as plt) com dados a partir do resultado da tabela verdade (Imagem 2).

## 6 DISCUSSÃO

As redes neurais convolucionais (CNNs) oferecem diversos benefícios no reconhecimento de danos em veículos, tornando-as uma ferramenta valiosa para aplicações de inspeção e avaliação de veículos. Dentre eles:

As CNNs são especialmente boas em detectar características complexas e sutis em imagens, o que é essencial ao avaliar danos em veículos. Elas podem identificar amassados, arranhões, rachaduras e outras imperfeições que podem ser difíceis de detectar manualmente.

As CNNs têm a capacidade de aprender automaticamente padrões e texturas associadas a diferentes tipos de danos. Isso significa que, à medida que mais dados são alimentados à rede, ela se torna melhor em reconhecer danos específicos, tornando-se adaptável a diferentes cenários e tipos de veículos.

Uma vez treinadas, as CNNs podem processar imagens de veículos rapidamente. Isso é particularmente importante em cenários como inspeções em massa em linhas de produção ou em pontos de entrada de veículos em estacionamentos, onde o tempo é crítico. Proporcionando eficiência e rapidez.

A automação proporcionada pelas CNNs ajuda a reduzir erros humanos na detecção de danos. As inspeções manuais podem ser suscetíveis a erros devido à fadiga ou distração, enquanto as CNNs mantêm um alto nível de consistência. As inspeções humanas também geram desgaste entre o inspetor e o inspecionado, a inspeção realizada por um sistema, tira a pessoalidade e paixões do ato.

A detecção precisa de danos em veículos é essencial para garantir a segurança dos motoristas e passageiros. Identificar danos que poderiam comprometer a integridade de um veículo ajuda a prevenir acidentes, melhorando a segurança

As CNNs podem ser implantadas em sistemas automatizados de grande escala, tornando-as ideais para aplicativos que envolvem uma grande quantidade de veículos, como sistemas de inspeção de segurança em aeroportos ou inspeções regulares em frotas de veículos. Tornando-a uma solução escalável.Ao fornecer uma avaliação objetiva e precisa dos danos, as CNNs ajudam na tomada de decisões sobre reparos, seguros ou substituição de veículos danificados.

As CNNs podem ser integradas com outras tecnologias, como sistemas de câmeras de vigilância e sistemas de processamento de imagem em tempo real, para criar sistemas de monitoramento e inspeção mais abrangentes.

No geral, as CNNs têm o potencial de aumentar a eficiência, a precisão e a segurança na detecção e avaliação de danos em veículos, tornando-se uma ferramenta valiosa para diversas indústrias, incluindo automobilística, seguros, logística e manutenção de frotas.

Enfim, possui diversos ramos de aplicação, o reconhecimento em danos em veículos por CNNs,, trará diversos benefícios para um mundo cada vez mais digital e conectado, aplicações como inspeção de veículos em locadoras, inspeção de sinistros para seguradoras, detecção de fraudes em fotos para confecção de seguro, inspeções regulamentadas pelo CONTRAN, com o propósito de garantir veículos seguros a serem postos em circulação.

## 7 CONCLUSÃO

As CNNs são uma classe especializada de redes neurais profundas que se destacam na análise de imagens devido à sua capacidade de aprender representações hierárquicas. Elas têm sido amplamente utilizadas no reconhecimento de objetos em imagens, tornando-as uma escolha natural para o desenvolvimento de soluções de inspeção de veículos.

Este artigo discute o uso de Redes Neurais Convolucionais (CNNs) no contexto do reconhecimento de danos em veículos, com um foco particular na integração do software chamado IVI (Inspeção Veicular Inteligente), desenvolvido com a finalidade de ser aplicado em diversos ramos, como como inspeção de veículos em locadoras, inspeção de sinistros para seguradoras, detecção de fraudes em fotos para confecção de seguro, inspeções regulamentadas pelo CONTRAN e diversas novas aplicações que podem vir a se adequar.

Nele podemos comprovar que o grau de assertividade de 93%, nas primeiras gerações e com um número baixo de imagens para treinamento, obteve grande sucesso, através do uso do método de Transfer Learning, onde iniciamos com um modelo com um conjunto de padrões e filtros pré-existentes que garantem uma boa assertividade mesmo com um conjunto pequeno de dados para

treinamento. Ao aumentarmos a quantidade de dados para treinamento, podemos esperar um grau de assertividade ainda maior, não ainda comparável a visão humana, pois, enquanto os modelos de CNN continuam a melhorar em tarefas de visão computacional, ainda há um longo caminho a percorrer antes que eles possam se aproximar verdadeiramente da capacidade de reconhecimento de imagens da visão humana em todos os cenários e nuances, sendo importante considerar que a visão humana é altamente dependente do contexto, da experiência e da compreensão semântica, algo que as redes neurais convolucionais ainda precisam evoluir para replicar completamente.

Porém o uso de CNNs, obtemos o resultado da tarefa de identificação sem os problemas que decorrem da interação humana, como conflitos e fraudes.

O potencial das CNNs no campo da inspeção veicular, para automatizar a detecção de danos em veículos, é grande, pode ser útil em várias aplicações, como inspeção de seguros e manutenção de frotas.

## REFERÊNCIAS


MIT XPro. 2020. Data Science and Big Data Analytics: Making Data-Driven Decisions.
https://globalalumni.xpromit.com/data-science-y-big-data-dsb-esp/?utm_campaign=mxp-dsb-esp&utm_source=MITxPRO&utm_medium=website

Coursera. 2019. Certificado Profissional IBM Data Science.
https://www.coursera.org/professional-certificates/ibm-data-science

O'Reilly. Artificial Intelligence Now - Current Perpectives from O'Reilly Media
O'Reilly Media, Inc.

Ruchi Chaturvedi, Sheetal Avirkar. Monitoring Road Condition using Neural Network
https://adasci.org/lattice-volume-4-issue-2/monitoring-road-condition-using-neural-network/

Iuliana Tabian, Hailing Fu, Zahra Sharif Khodaei. 2019.
A Convolutional Neural Network for Impact Detection and Characterization of Complex Composite Structures.
https://www.mdpi.com/1424-8220/19/22/4933

Git Hub
GitHub - fchollet/deep-learning-models: Keras code and weights files for popular deep learning models.

Vehicle inspection in car rental
https://www.carrentalgateway.com/glossary/vehicle-inspection/