SEVEN
publicações acadêmicas

Applications of Convolutional Neural Networks in vehicle damage recognition

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

Erick Roberto Furst Brito

ABSTRACT
Convolutional Neural Networks (CNNs) have demonstrated exceptional performance in a variety of computer vision tasks. This article explores the use of CNNs in the context of vehicle damage recognition, with a focus on the integration of a product called IVI (Intelligent Vehicle Inspection) developed for this purpose.

Keywords: CNN, Convolutional Neural Networks, Computer Vision, Vehicle Inspection.

1 INTRODUCTION

Vehicle inspection at a car rental company, for example, is an important process that aims to ensure the safety, quality, and compliance of each rental vehicle before and after they are delivered to customers. The inspection process can vary from one rental company to another, but generally includes the following steps:

Vehicle Check-in; Damage Registration; Mileage Logging; Fuel Level Check; Documentation Verification and Basic Cleaning and Maintenance.

The vehicle inspection process at rental companies is critical to ensuring the safety of customers, protecting the rental company's assets, and maintaining customer satisfaction. Any additional damage caused by the customer during the rental period is also recorded and may result in additional costs.

For the damage check stage, sometimes there can be disagreements regarding liability, many rental companies currently already automate most of the process, leaving the customer almost in a self-service situation, were it not for the verification of the damage. Computer vision through Convolutional Neural Networks may be the answer to this.

Convolutional neural networks are a specialized subset of deep neural networks that have excelled in image analysis due to their ability to learn hierarchical representations. In particular, they have been widely applied in the recognition of objects in images, making them a natural choice for the development of vehicle inspection solutions.

Convolutional neural networks (CNNs) are a type of neural network architecture designed specifically for computer vision tasks, such as pattern recognition in images and videos. They are called "convolutional" because of the fundamental operation called convolution, which plays a central role in this type of network.

CNNs receive an image as input. Each image is represented as an array of pixels, where each pixel has values that represent its color intensity (for example, grayscale pixel values or RGB channel values for color images).

The convolution operation is applied at the initial layer of the network. Convolution involves a small array called a filter or kernel, which slides across the input image to extract local resources. As the filter moves through the image, it performs multiplication and addition operations to create a feature map.

The resulting feature maps capture relevant information from the image, such as borders, textures, and simple patterns. After convolutional layers, it is common to use pooling layers to reduce the dimensionality of feature maps and extract more important information. The pooling layer (usually max-pooling) reduces the spatial resolution while retaining the most important characteristics.

After several convolutional and pooling layers, CNNs often have fully connected layers, similar to traditional neural networks. These layers learn to combine the previously extracted information to accomplish the final grading or other desired task.

The last layer of CNN is the output layer, which depends on the specific task the network is solving. For example, in an image classification problem, the output layer might have a neuron for each possible class, with activation functions such as softmax to produce probabilities of each class.

During training, CNN adjusts its parameters (filter weights, connection weights in the fully connected layers) using optimization algorithms, such as gradient descent, to minimize a loss function that evaluates how well the network is doing its task.

Once the training is complete, CNN can be used to classify new images or perform other tasks related to computer vision. The input image passes through the network, and the output layer provides the desired predictions or outcomes.

CNNs are highly effective in computer vision tasks due to their ability to automatically learn relevant features at different levels of abstraction. They have revolutionized fields such as image recognition, object detection, image segmentation, and more. In addition, advanced architectures such as residual networks (ResNets) and deep convolutional neural networks (deep CNNs) have made it possible to train deeper, more powerful networks for complex tasks.

Convolutional Neural Networks are based on two main operations: convolution and clustering.

Convolution is a mathematical operation used to process an image. Imagine that you have an input image (an array of pixels) and a set of filters (kernels) that are smaller arrays. Convolution consists of sliding these filters over the input image and performing multiplication and addition to obtain a feature map. This can be expressed mathematically as follows for a single filter:

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

Where:

- I is the input image.
- K is the filter (kernel).
- H and W are the dimensions of the filter.

Once the convolution is done, it is common to apply a grouping operation (Pooling), usually the maximum grouping (max-pooling). This operation reduces the size of the output when selecting the maximum value in a specific region. For example, if you use a 2x2 window, you will get the maximum value of each group of 4 values in the convolution output.

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

These operations are repeated in successive layers of CNN to extract progressively more abstract features from the image. These traits are then fed into a fully connected neural network for final classification.

It is important to mention that in a full CNN, multiple filters are used on each convolutional layer to capture different features of the image, and convolution and grouping operations are performed in parallel across multiple color channels if the image is color (RGB).

2 METHODOLOGY

2.1 CNN ARCHITECTURE

A model was created in the Python 3.5 programming language, using the Keras library. The weight used was downloaded from a Git Hub page, with pre-trained models (https://github.com/fchollet/deep-learning-models/releases/) for Keras, Tensorflow. VGG16 is a popular convolutional neural network architecture for computer vision tasks. It is known for its relative simplicity and good performance in image classification tasks. The initial model simply identifies a damaged vehicle from an undamaged one, and later evolved into a model identifying the types of damage presented, for the purposes of simplicity and chronology, I will present the main excerpts of the code for differentiating from damaged to undamaged vehicle.

The model initially created using the pre-trained VGG16 architecture:

Code:

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

Pre-trained VGG16 weights were used that were trained on the image classification task in the ImageNet dataset. Using pre-trained weights is a common practice when training computer vision models, as these weights contain useful information learned from a large dataset.

It was also decided not to include the dense (fully connected) layer on top of the neural network. Instead, I used VGG16's architecture down to the last convolutional layer. This is useful when we want to customize the network for a specific task, such as transfer learning.

The pooling layer to be used after the convolutional layers has been set to "max", which means that the pooling layer will use the maximum operation to reduce the spatial dimensions of the convolutional features.

In summary, the code creates a VGG16 base model with pre-trained ImageNet weights, removes the top dense layer, defines the format of the input images, and specifies the use of the maximum pooling layer after the convolutional layers. This template can be used as a basis for learning transfer tasks, where you can add your own custom layers on top to train you on a specific task, such as image classification.

Code:

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

Disabling training of the base model layers, so that only the layers added later (custom layers) are trained during the global network training process. This is particularly useful when you have a smaller dataset or an image classification task that is related to what the base model has already learned. Freezing the layers of the base model helps preserve the resources and knowledge it has already acquired, while allowing you to tailor the model for your specific task.

The reason for doing this is to use the "Transfer Learning" method. The idea behind the transfer of learning is that you can use a pre-trained model, such as VGG16 with ImageNet weights, which have already been trained with a solid database that has already learned how to extract useful features from images. Instead of training the entire model from scratch (which can be computationally expensive), you can freeze the base model layers so they don't change, and then add custom layers on top to learn the specific task you want.

Code:

base_model.summary()

The base model generated:

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 TRAINING

This step consisted of training a binary classification model (i.e., a model that classifies samples into two classes (damaged or undamaged). Initially, you load the bottleneck characteristics

(intermediate representations) of the training images that were previously calculated and saved in .npy files. These characteristics will serve as input to the classification model.

The numpy array "train_labels" contains class labels for the training samples. These labels are created based on the number of positive and negative samples, where 0 represents "Harm" and 1 "No Harm". The numpy array "val_labels" contains class labels for the validation samples, following a pattern similar to that of the training samples.

A sequential neural network model was created with an input layer (Flatten), a dense layer (Dense) with ReLU activation, a dropout layer (Dropout) and a dense output layer with sigmoid activation.

The build configured the model for training, specifying the optimizer, the loss/loss function (binary_crossentropy for binary sorting), and the metrics to monitor (accuracy).

The checkpoint defined a callback that served to monitor the validation accuracy metric (val_acc) and save the model weights whenever validation accuracy improves. This helps prevent overfitting and allows for model tracking during training. Training takes place by the number of epochs specified per nb_epoch.

The "Lr" parameter refers to the learning rate of the SGD (Stochastic Gradient Descent) optimizer used to train the neural network. The learning rate is a crucial hyperparameter in training neural networks and controls the size of the steps the optimizer takes when adjusting the network weights during training. A lower learning rate, such as 0.0001, means that network weights are adjusted with small steps with each training iteration. This can help model convergence, especially when training on complex datasets. However, choosing the appropriate learning rate can be challenging and often require empirical adjustments to get the best performance out of the model.

It's important to note that learning rate is just one of the hyperparameters that affect neural network training, and finding the right combination of hyperparameters for a specific problem is a critical part of developing effective machine learning models.

Finally, the function returns the trained model and training history (fit.history), which contains information about the training metrics over the epochs.

Code:

```
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 to match the size of the 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 TRAINING EXECUTION

Training and storage of results for later evaluation. The "var_modelo" variable will contain the trained model and we will use it to make predictions, and the "var_history" variable to analyze the model's performance over the training epochs.

Code:

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

The training:

```
(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 DATASET

In this section, we will describe in detail the data collection process to train the damaged vehicle identification model. The dataset plays a key role in the development of any machine learning system, and it is crucial to document how samples were obtained to ensure transparency and reproducibility of the experiment.

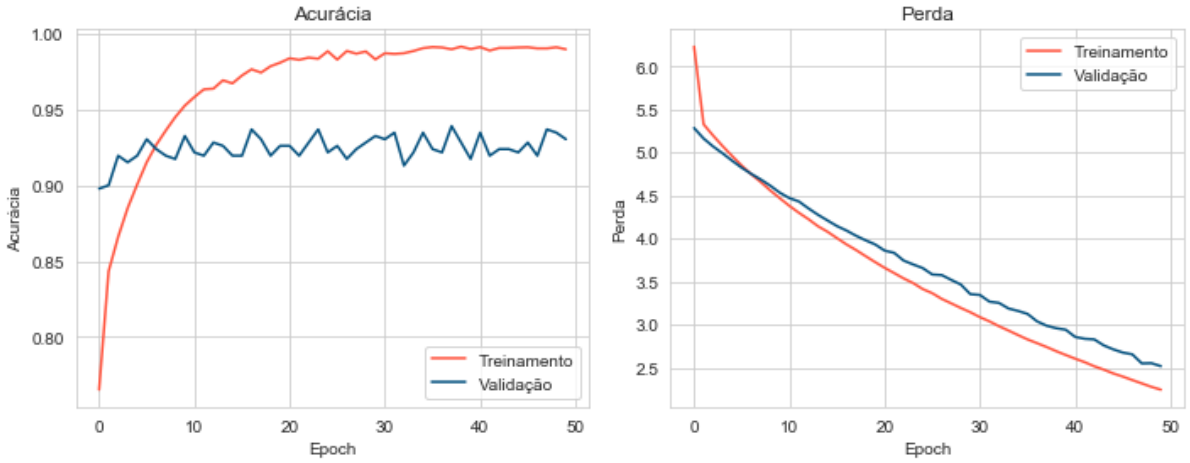
A total of 4,351 (four thousand three hundred and fifty-one) images of damaged vehicles were obtained through searches on the Google search site, and a few hundred of this sample were obtained through photos taken from a cell phone in a yard of a car rental company and 2,536 (two thousand, five hundred and thirty-six) images of undamaged vehicles obtained in the same way. These images were separated into folders, manually, by me. There are two folders, one named "Damage" and the other named "No Damage".

The "Damage" folder contained a total size of 503 megabytes (527,775,421 bytes) and the "Damage-Free" folder a total size of 376 megabytes (395,072,008 bytes).

5 RESULTS

The model with the best accuracy of 93% (0.939139425453186) was obtained in the 38th season, with a loss of 2.98% (2.9858194682909094) - Image 1 -. For validation, 230 (Two hundred and thirty) images were used. Of these 230 images, 6.5% were only of scratches without any other type of damage, of which none were misclassified.

Image 1



Source: Survey Data

Image generated by the application of plot function (plot, import matplotlib.pyplot as plt) with data from the variable "var_history" filled in during the training phase.

The graph on the left displays the accuracy rates through the epochs of training.

The graph on the right displays the Loss Rate through the Epochs of training.

The truth table (Image 2) after fine tuning presented the following results, with the validation data:

Image 2

Source: Survey Data

Image generated by applying the heatmap function (sns.heatmap, import seaborn as sns), from the model generated against validation data.

The generated validation graph shows the following results, in a validation sample of 460 images:

True Positives: 213

False Positives: 17

False negatives: 15

True Negatives: 215

Summarizing the confusion table on the ROC curve:

True Positive = Sensitivity

False positive = Specificity

$$True\ Positive\ rate = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Where the "True Positive Rate" tells you the proportion of correctly classified "Damaged Cars" samples.

$$False\ Positive\ Rate = 1 - Specificity = \frac{False\ Positive}{False\ Positive + True\ Negative}$$

Where the "False Positive Rate" tells you the proportion of samples of "No Damage Cars" incorrectly classified.

The ROC curve, or "Receiver Operating Characteristic Curve," is a graphical tool used to evaluate the performance of binary classifiers such as machine learning models, especially on binary classification problems (two classes, usually positive and negative).

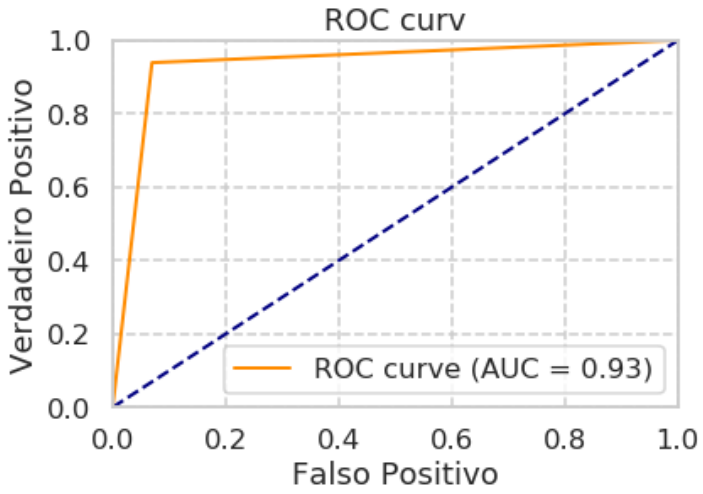
The ROC Curve represents the relationship between the True Positive Rate (TPR) and the False Positive Rate (FPR) at different thresholds for a model's predictions. An ideal ROC Curve is closer to the top left corner of the graph, where the TPR is high and the FPR is low. The more the ROC Curve deviates from this diagonal line, the better the model performs. The value of the area under the ROC Curve (AUC-ROC) is also commonly calculated to quantify the overall performance of the model, where a value of 1 indicates perfect performance and a value of 0.5 indicates random performance (no discrimination between classes).

The blue diagonal line indicates that the True Positive Rate is equal to the False Positive Rate, i.e., the model next to the blue line is irrelevant, as it would always present the same proportion of classifications with hits and misses.

That is, the farther away the diagonal line, the better.

The model presented a result of 0.93 (Image 3), a model considered excellent, with the ROC curve as close to 1 (one) as possible.

Image 3



Source: Survey Data

Image generated by the application of plot function (plot, import matplotlib.pyplot as plt) with data from the result of the truth table (Figure 2).

6 DISCUSSION

Convolutional neural networks (CNNs) offer several benefits in recognizing vehicle damage, making them a valuable tool for vehicle inspection and evaluation applications. Among them:

CNNs are especially good at detecting complex and subtle features in images, which is essential when assessing vehicle damage. They can identify dents, scratches, cracks, and other imperfections that can be difficult to detect manually.

CNNs have the ability to automatically learn patterns and textures associated with different types of damage. This means that as more data is fed into the network, it becomes better at recognizing specific damages, making it adaptable to different scenarios and vehicle types.

Once trained, CNNs can process vehicle footage quickly. This is particularly important in scenarios such as mass inspections on production lines or at vehicle entry points in parking lots, where time is critical. Providing efficiency and speed.

The automation provided by CNNs helps reduce human error in detecting damage. Manual inspections can be susceptible to errors due to fatigue or distraction, while CNNs maintain a high level of consistency. Human inspections also generate wear and tear between the inspector and the inspected, the inspection carried out by a system, takes away the personhood and passions of the act.

Accurate vehicle damage detection is essential to ensure the safety of drivers and passengers. Identifying damage that could compromise the integrity of a vehicle helps prevent accidents, improving safety

CNNs can be deployed in large-scale automated systems, making them ideal for applications involving a large number of vehicles, such as security screening systems at airports or regular inspections of vehicle fleets. Making it a scalable solution. By providing an objective and accurate assessment of damage, CNNs help in making decisions about repairs, insurance, or replacement of damaged vehicles.

CNNs can be integrated with other technologies, such as surveillance camera systems and real-time image processing systems, to create more comprehensive monitoring and inspection systems.

Overall, CNNs have the potential to increase efficiency, accuracy, and safety in detecting and assessing vehicle damage, making them a valuable tool for various industries, including automotive, insurance, logistics, and fleet maintenance.

Finally, it has several branches of application, the recognition of vehicle damage by CNNs, will bring several benefits to an increasingly digital and connected world, applications such as vehicle inspection in rental companies, claims inspection for insurance companies, detection of fraud in photos for insurance making, inspections regulated by CONTRAN, with the purpose of ensuring safe vehicles to be put into circulation.

7 CONCLUSION

CNNs are a specialized class of deep neural networks that excel in image analysis due to their ability to learn hierarchical representations. They have been widely used in recognizing objects in images, making them a natural choice for the development of vehicle inspection solutions.

This article discusses the use of Convolutional Neural Networks (CNNs) in the context of vehicle damage recognition, with a particular focus on the integration of software called IVI (Intelligent Vehicle Inspection), developed with the purpose of being applied in various branches, such as vehicle inspection in rental companies, claims inspection for insurance companies, photo fraud detection for insurance making, inspections regulated by CONTRAN and several new applications that may be suitable.

In it we can prove that the degree of assertiveness of 93%, in the first generations and with a low number of images for training, was very successful, through the use of the Transfer Learning method, where we started with a model with a set of pre-existing patterns and filters that guarantee good assertiveness even with a small set of data for training. By increasing the amount of data for training, we can expect an even greater degree of assertiveness, not yet comparable to human vision, because while CNN models continue to improve in computer vision tasks, there is still a long way to go before they can truly approach the image recognition capability of human vision in all scenarios and nuances. It is important to consider that human vision is highly dependent on context, experience, and semantic understanding, something that convolutional neural networks have yet to evolve to fully replicate.

However, with the use of CNNs, we obtain the result of the identification task without the problems that arise from human interaction, such as conflicts and fraud.

The potential of CNNs in the field of vehicle inspection, to automate the detection of damage to vehicles, is great, it can be useful in various applications, such as insurance inspection and fleet maintenance.

REFERENCES

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/