UNIVERSIDADE PRESBITERIANA MACKENZIE
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA
TECNOLOGIA EM CIÊNCIAS DE DADOS

PROJETO APLICADO II

Projeto Sentinela: Automatização do Controle de Acesso Veicular em Condomínios.

PROFESSOR: FELIPE ALBINO DOS SANTOS

GRUPO:

ALINE CORRÊA – 10414773 – 10414773@mackenzista.com.br
ISAQUE PIMENTEL – 10415608 – 10415608@mackenzista.com.br
MAIKI SOARES – 10415481 – 10415481@mackenzista.com.br
VANESSA CORDEIRO – 10415118 – 10415118@mackenzista.com.br

São Paulo
2024

SUMÁRIO

1. CONSOLIDAÇÃO DOS RESULTADOS DO MÉTODO ANALÍTICO

Nessa etapa, foi realizada o processamento dos dados; separação entre dados de treinamento e de teste; treinamento do modelo.

1.1. PROCESSAMENTO DOS DADOS

A leitura dos dados é uma etapa essencial. Durante este processo, todas as imagens das placas veiculares são processadas e convertidas em arrays numpy utilizando a biblioteca *OpenCV.* As imagens são redimensionadas para um tamanho fixo 224 x 224, que é o tamanho padrão compatível com modelo pré-treinado que será utilizado.

Além disso, as imagens será renormalizadas para o formato de 8 bits, dividindo-se o array numpy por 255. As etiquetas das imagens também são renormalizadas, uma vez que a saída do modelo pré-treinado varia entre 0 e 1. Esse processo é implementado no arquivo *read_data.py*.

Vale ressaltar que cada imagem é representada por um array numpy tridimensional de tamanho 224 x 224 x 3, enquanto cada etiqueta é representada por um array numpy unidimensional de tamanho 4. A coleção de imagens e etiquetas é salva no formato .npy (a saber, *X.npy* e *y.npy*) para serem reutilizadas em outras etapas do processo de aprendizagem.

1.2. TREINAMENTO DO MODELO

Após repartir as imagens e etiquetas (representadas pelas variáveis X e y) em conjuntos de treinamento e teste utilizando a biblioteca *sklearn*, com uma proporção de 4:1 de dados de treinamento para dados de teste, obtemos:

```
Formato de X_train: (346, 224, 224, 3)
Formato de X_test: (87, 224, 224, 3)
Formato de y_train: (346, 4)
Formato de y_test: (87, 4)
```

Figura 1 – Tamanho dos arrays numpy representado os conjuntos de treinamento e de teste.
Fonte: Elaboração própria (2024).

O modelo pré-treinado de Deep Learning escolhido é o **Inception-ResNet-v2**. Esse é um modelo de rede neutral convolucional (CNN) que já foi treinado com milhares de imagens da base de dados *ImageNet*. Essa rede possui um total de 164 camadas sendo classificar imagens em mais de 1000 categorias de objetivos, como, caneta, teclado e outros. Portanto, essa rede possui uma boa representação de uma multitude de objetos.

Segue a arquitetura da rede neural abaixo:

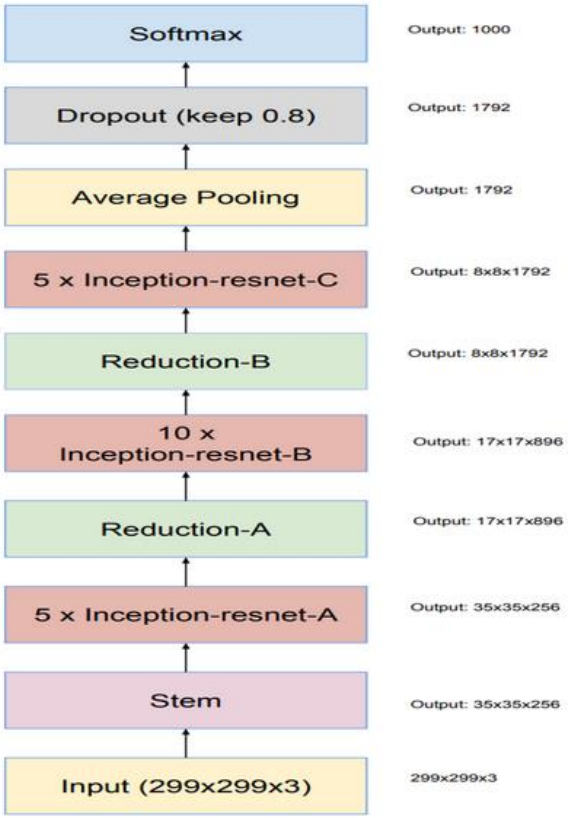
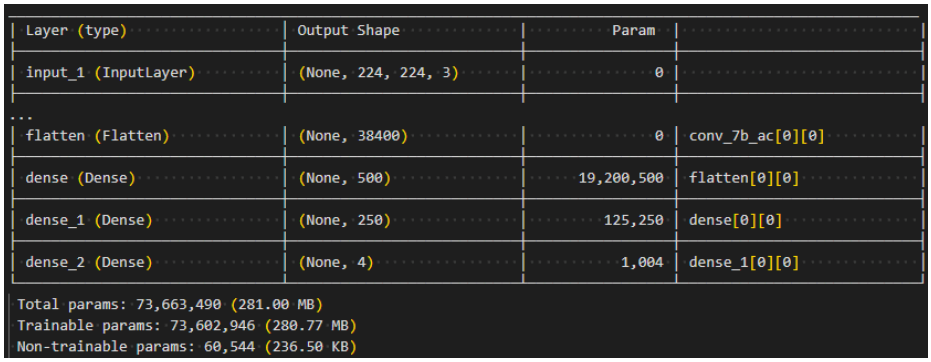


Figura 2 – Arquitetura do modelo Inception-ResNet-v2 com aproxidamente 164 camadas internas. Fonte: Inception-ResNet-v2 by Szegedy et al..

Assim, usaremos o modelo **Inception-ResNet-v2** com peso pré-treinados para treiná-los ao nosso banco de dados. Para isso, importaremos e ajustaremos as dimensões das entradas e saídas utilizando a biblioteca *TensorFlow*. É importante mencionar que a saída original do modelo foi redimensionada para comportar a saída das etiquetas (um array numpy unidimensional de tamanho 4). Posteriormente, o modelo foi compilado usando uma função de perda é **MSE** (i.e., *Mean Squared Error*) e o otimizador Adam com uma taxa de aprendizagem de **0.0001**.



Layer (type)	Output Shape	Param	
input_1 (InputLayer)	(None, 224, 224, 3)	0	
...			
flatten (Flatten)	(None, 38400)	0	conv_7b_ac[0][0]
dense (Dense)	(None, 500)	19,200,500	flatten[0][0]
dense_1 (Dense)	(None, 250)	125,250	dense[0][0]
dense_2 (Dense)	(None, 4)	1,004	dense_1[0][0]

Total params: 73,663,490 (281.00 MB)
Trainable params: 73,602,946 (280.77 MB)
Non-trainable params: 60,544 (236.50 KB)

Figura 3 – Resumo do modelo **Inception-ResNet-v2** modificado. Output original: 38400. Output modificado: 4.
Fonte: Elaboração própria (2024).

A etapa de treinamento foi realizada em um computador com *Windows 11 Home Single Language*, equipado com um processador *12th Gen Intel(R) Core(TM) i5-12500T 2.00 GHz* e 7,70 GB de RAM utilizável. Devido às limitações de recursos computacionais, treinamos o modelo com os dados X_train e y_train usando um tamanho de lote (*batch size)* de 16 em apenas 8 *epochs*.

Após o treinamento, o modelo foi salvo no arquivo *object_detection.keras*. Esse processo de re-treinamento foi implementado no arquivo *train_model.py*.

2. REVISÃO DO DESEMPENHO DO MÉTODO ANALÍTICO

Nessa etapa, foi realizada a avaliação do desempenho do modelo sobre os dados de treinamento e validação.

Conforme explicado na etapa de treinamento, o critério de otimização do modelo **Inception-ResNet-v2** foi o erro quadrático médio (MSE em inglês), em vez da acurácia. Na verdade, o algoritmo de optimização visa minimizar uma função de erro entre as etiquetas de teste/treinamento e as etiquetas previstas pelo modelo. Como as etiquetas são representa por arrays numpy unidimensionais de tamanho 4, elas podem ser visualizadas em um espaço euclidiano de dimensão 4, onde o erro é simplesmente que a distância euclidiana entre esses dois pontos: observado e o previsto.

Utilizando a ferramenta TensorBoard, executamos o comando `tensorboard --logdir="./object_detection` para obter o histórico de treinamento. Abaixo, podemos observar os valores da função de perda (*epoch_loss*) para cada época:

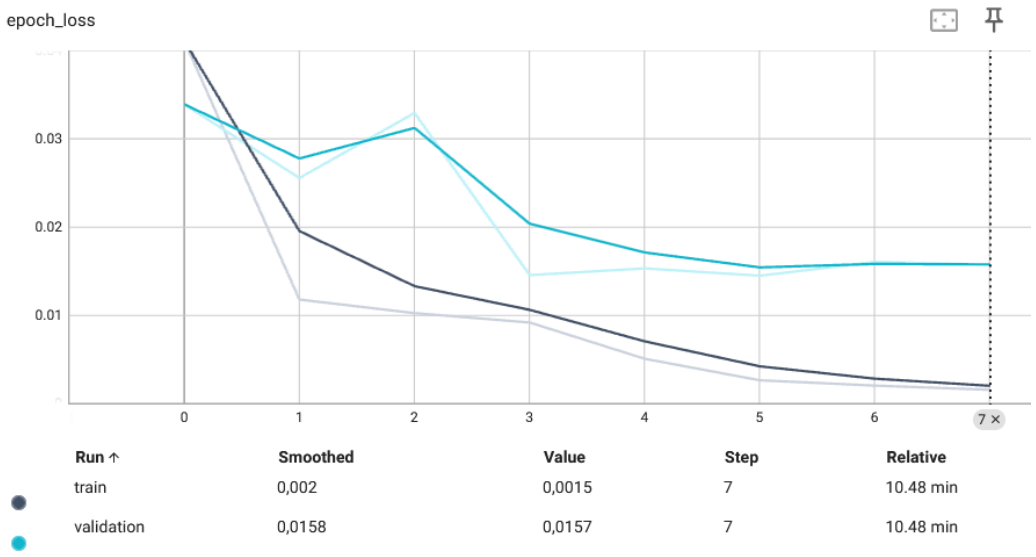


Figura 4 – Curva de aprendizagem (i.e., valor da *loss function*) para cada *epoch* nos dados de treinamento e de validação**.**
Fonte: Elaboração própria (2024).

De modo geral, o valor da *loss* diminui com o passar épocas, indicando que o modelo está aprendendo a identificar as placas veiculares cada vez melhor no *dataset* de treinamento. Certamente, aumentar o número de *epochs* resultaria em um melhor score para o modelo ao final do treinamento.

De mesma forma, o valor da *loss* decresce com a *epoch* para o *dataset* de validação. Observamos que a *loss* para os dados de validação é maior, justamente porque eles não são usados para treinar o modelo, mas sim para avaliar sua capacidade de generalização para dados não vistos anteriormente. Como o valor da *loss* dos dados de treinamento ainda não se estabilizou, o processo de aprendizagem não deveria ter sido encerrado em 8 *epochs*.

Agora, com o modelo treinado e salvo, avançaremos para a detecção de placas e reconhecimento de caracteres propriamente ditos em uma imagem de teste.

3. DESCRIÇÃO DE RESULTADOS PRELIMINARES

Nessa etapa, foi realizada a predição do modelo, ou seja, a deteção das placas veiculares juntamente com o reconhecimento de caracteres das placas.

3.1. DETECÇÃO DE PLACAS VEICULARES.

Carregando o modelo previamente salvo no arquivo *object_detection.keras*, somos capazes de utilizá-lo para fazer previsões sobre uma nova imagem de carro, identificando a localização da placa veicular. Para isso, obtivemos novas imagens de testes que não foram utilizadas no processo de treinamento e validação.

Nosso projeto de detecção automática de placas veiculares foi originalmente concebido para detecção de placas brasileiras. Após uma busca por imagens na internet, encontramos as seguintes imagens que servirão como conjunto de teste:

Figura 5 – Amostra das imagens de veículos com placas brasileiras incluindo o modelo do Mercosul. Fonte: Busca por imagens “Foto de frente de carros com placas brasileiras”.

As imagens de teste foram carregadas da pasta “dataset/TEST” e processadas da mesma maneira que as imagens de treinamento/validação. Usando o modelo, foram previstas as quatro coordenadas (x_min, x_max, y_min, y_max) identificando a localização das placas. Em seguida, as coordenadas foram transformadas de valores entre 0 e 1 para valores correspondentes ao tamanho das imagens originais. Esse processo de deteção de placas foi implementado no arquivo *detect_plate.py*.

Figura 6 – Amostra das imagens de veículos com placas identificadas por uma caixa amarela.
Fonte: Elaboração própria (2024).

De modo geral, as placas veiculares foram identificadas nas imagens de teste. Observamos que a imagem com mais zoom na placa obteve a melhor detecção, enquanto os veículos fotografados de frente tiveram suas placas mais bem classificadas do que veículos fotografados de trás. Seria interessante testar a detecção para veículos fotografados em ângulo ou com baixa visibilidade (pouca iluminação ou qualidade ruim da imagem).

3.2. RECONHECIMENTO DE PLACAS VEICULARES.

As imagens das placas detectadas para os veículos de teste foram salvas na pasta “dataset/TEST”, depois reprocessadas para a realização do OCR (Optical Character Recognization). Utilizando o software chamado Tesseract OCR e sua API em Python (pytesseract), foi realizado o reconhecimento de caracteres nas placas detectadas.

O Tesseract é uma ferramenta *open source* que realiza as tarefas de segmentação do texto do plano de fundo quanto melhor for a qualidade da imagem. Assim, se a imagem apresentar algum ruído, iluminação ruim ou baixo contraste, a tarefa de reconhecimento será prejudicada. Apesar de ser uma ferramenta robusta, possui limitações, como não reconhecer bem imagens sob perspectiva distorcida ou fundo complexo, podendo gerar lixo no output. Esse processo de reconhecimento de caracteres a partir das placas identificadas foi implementado no arquivo *recognize_plate.py*.

String vazia

“BEESR22)|\n”

“AE\n”

String vazia

Figura 7 – Amostra das imagens de veículos com placas reconhecidas pelo Tesseract OCR.
Fonte: Elaboração própria (2024).

É claro que não reconhecemos as placas corretamente. Para a terceira placa, temos 80% das informações, com um caractere mal identificado e um tanto de lixo adicionado. Para melhorar o processo de reconhecimento, as placas precisam ser mais bem identificadas. Assim é recomendado aumento o número de *epochs* a fim que o erro de detecção seja o mínimo possível e a maior fonte de erro será no algoritmo do Tesseract sobre o qual não temos muito o que fazer.

De maneira geral, com poucas *epochs*, esse modelo pode ter um desempenho inferior especialmente se consideramos todas as etapas do processo. No entanto, essa primeira iteração permitiu entender as áreas de melhoria para nosso modelo, como, o aumento do número de *epochs* e a inclusão de mais dados de treinamento. Tentaremos implementar essas melhorias para a próxima etapa do projeto.

3.3. MODELO DE NEGÓGIO

Nosso modelo de negócio consiste em um sistema de detecção e reconhecimento de placas chamado Sentinela, solicitado por uma administradora de

condomínios que planeja implementá-lo nos condomínios de casas em São José dos Campos.

Para desenvolver o sistema de reconhecimento de placas veiculares, utilizamos diferentes bibliotecas de *machine learning* e uma base de dados contendo imagens de carros com placas estrangeiras. No entanto, nosso *dataset* era bastante limitado contendo apenas 433 imagens. Com base nos resultados preliminares, é evidente que precisamos investir em um *dataset* maior e mais abrangente.

Além disso, nosso modelo requer um treinamento mais aprofundado, que pode ser alcançado utilizando GPUs para acelerar os cálculos durante o treinamento.

4. ELABORAÇÃO DO ESBOÇO DO STORYTELLING

Com os elementos desenvolvidos no projeto, foi feito um esboço do storytelling a ser feito em uma eventual apresentação de negócio.

> *“Durante uma disciplina de Ciência de Dados, um grupo de alunos, cheios de curiosidade, paixão por novas tecnologias e muita vontade de empreender, se lançou num projeto desafiador: criar uma solução inovadora para um problema real com aplicação comercial. Assim nasceu o Sentinela, um sistema automático de detecção e reconhecimento de placas veiculares a partir de imagens de veículos.*
>
> *A jornada começou com a concepção do Sentinela, um projeto de Aprendizado de Máquina destinado a revolucionar a forma como identificamos veículos. Fomos abordados por uma administradora de condomínios, que via no Sentinela a solução ideal para garantir a segurança e o controle de acesso em propriedades administradas.*
>
> *Nosso modelo de negócio estava traçado: oferecer um sistema de detecção e reconhecimento de placas de excelência que atendesse às necessidades de condomínios em São José dos Campos - SP. Para alcançar nossa meta, mergulhamos de cabeça no vasto mundo da aprendizagem de máquina, utilizando diferentes bibliotecas em python e ferramentas para desenvolver nosso primeiro modelo.*

No entanto, logo nos deparamos com um desafio significativo: nosso conjunto de dados era limitado, contendo apenas algumas centenas de imagens de carros com placas estrangeiras. Para criar um modelo robusto e preciso, precisávamos de mais dados e poder computacional para treinar nosso algoritmo de maneira mais profunda e abrangente.

Optamos por utilizar o modelo pré-treinado Inception-ResNet-v2, uma poderosa rede neural convolucional treinada com milhares de imagens da base de dados ImageNet. Adaptamos esse modelo às nossas necessidades e o treinamos com nosso conjunto de dados, ajustando parâmetros e otimizando o processo de aprendizado.

Apesar de nossos esforços, os resultados preliminares não foram tão promissores quanto esperávamos. Reconhecemos que nossas placas ainda não eram identificadas corretamente, com erros significativos em algumas delas. Percebemos que precisávamos de mais tempo de treinamento e mais dados para melhorar a precisão do nosso modelo.

No entanto, essa primeira iteração do Sentinela nos proporcionou valiosas lições. Compreendemos a importância de aumentar o número de epochs e a quantidade de dados de treinamento para aprimorar nosso modelo. Reconhecemos os desafios que ainda temos pela frente, mas estamos determinados a superá-los e tornar o Sentinela uma solução confiável e eficaz para nossos clientes.

Assim, nossa caminhada continua, impulsionada pela paixão pela inovação e pelo desejo de criar um impacto positivo no mundo. Com cada iteração, estamos mais perto de alcançar nossa meta e transformar o Sentinela em uma ferramenta indispensável para a segurança e o controle de acesso em condomínios de todo o país".