UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

FRANCISCO DELMAR KURPIEL

LOCALIZAÇÃO DE PLACAS VEICULARES EM VÍDEO USANDO REDES NEURAIS CONVOLUCIONAIS PROFUNDAS

TRABALHO DE CONCLUSÃO DE CURSO

CURITIBA

2016

FRANCISCO DELMAR KURPIEL

LOCALIZAÇÃO DE PLACAS VEICULARES EM VÍDEO USANDO REDES NEURAIS CONVOLUCIONAIS PROFUNDAS

Trabalho de Conclusão de Curso apresentado aos Departamentos Acadêmicos de Informática e Eletrônica como requisito parcial para obtenção do grau de Bacharel em Engenharia de Computação da Universidade Tecnológica Federal do Paraná .

Orientador: Prof. Dr. Bogdan Tomoyuki Nassu

CURITIBA

2016

Para o meu filho Luka.

AGRADECIMENTOS

Ofereço a minha mais sincera gratidão:

ao orientador, Prof. Dr. Bogdan Tomoyuki Nassu pelo estímulo, paciência e principalmente pela confiança - eu não teria terminado sem essas três coisas;

à minha esposa Luciane, que a grande custo permitiu que eu me dedicasse exclusivamente a este trabalho;

à Leonardo Cooper, que entendeu a importância deste trabalho para mim, e encontrou uma forma de permitir que eu me concentrasse nele por várias semanas em período integral;

À minha mãe Laurecy, que sempre me apoiou e acreditou em mim.

RESUMO

Kurpiel, Francisco Delmar. Localização de Placas Veiculares em Vídeo Usando Redes Neurais Convolucionais Profundas. 74 f. Trabalho de Conclusão de Curso – Bacharelado em Engenharia da Computação, Universidade Tecnológica Federal do Paraná. Curitiba, 2016.

Localização de placas veiculares é um componente importante de sistemas de controle e fiscalização de tráfego contemporâneos. Alguns sistemas incluem localização como parte fundamental de um subcomponente. Em alguns casos o desempenho do sistema de localização pode definir a viabilizada do sistema como um todo. Neste trabalho, é proposto um método de localização aproximado de placas veiculares em vídeo desenvolvida usando redes neurais convolucionais aplicadas diretamente aos valores RGB dos *pixels*. Na implementação aqui exposta a placa pode ser localizada com uma margem de erro de $\pm 20 \times \pm 60$ *pixels*. O treinamento é feito aplicando distorções às imagens, como ruído e variação aleatória de brilho, contraste e saturação, para produzir um modelo robusto, mesmo sem uso de filtros ou normalização. A proposta inclui um método eficiente de particionar a imagem de forma a reduzir consideravelmente o número de vezes que a rede neural precisa ser aplicada. Os indicadores *precision* e *recall* obtidos foram, respectivamente, 98,9% e 96,7% enquanto cada quadro de 480×768 é processado em 67,7 ms em um notebook Core i5 com GPU GTX 750M.

Palavras-chave: Localização de Placas Veiculares, Redes Neurais Convolucionais

ABSTRACT

Kurpiel, Francisco Delmar. License Plate Localization From Video Using Convolutional Neural Networks. 74 f. Trabalho de Conclusão de Curso – Bacharelado em Engenharia da Computação, Universidade Tecnológica Federal do Paraná. Curitiba, 2016.

License plate extraction is an important component of contemporary's urban traffic control and surveillance systems. Some systems include extraction as an important part of one of it's subcomponents. In some cases the extraction performance characteristics may be determinant of the whole system viability. This is a proposal for a method that obtains the approximate location of vehicle's license plates in a video using convolutional neural networks applied directly to the value of the RGB pixels. In the implementation being presented the vehicle's license plate position can be determined with an error margin smaller than $\pm 20 \times \pm 60$ pixels. During the training, some random distortions are added to the images, such as noise and changing the image's brightness, contrast and saturation, to produce a robust model, even when used without any filter or normalization. This proposal includes a method for efficient image partitioning, in such a way that the number of times the neural network has to be applied is considerably reduced. Measured precision was 98.9% and recall was 96.7% for locating license plates while processing each 480×768 frame in 67.6ms on a Core i5 notebook with a GTX 750M GPU.

Keywords: License Plate Localization, Convolutional Neural Networks

LISTA DE FIGURAS

Figura 1 – Resultado esperado 11
Figura 2 – Exemplo de rede neural totalmente conectada 14
Figura 3 – Comparação da tangente hiperbólica com a ReLu 15
Figura 4 – Exemplo de correlação cruzada 1D 17
Figura 5 – Exemplo de convolução 2D 20
Figura 6 – Convolução com múltiplos filtros e entrada em lote 22
Figura 7 – Exemplo de *maxpool* 2×2 22
Figura 8 – Exemplo de rede neural convolucional completa 24
Figura 9 – Convolução sendo aplicada em uma imagem RGB 28
Figura 10 – Exemplo de rede neural convolucional completa aplicada à uma imagem RGB 28
Figura 11 – Exemplo de Classificador Logístico 30
Figura 12 – Etapas do método proposto para localização de placas veiculares 36
Figura 13 – Exemplo de três tamanhos de partições 37
Figura 14 – Exemplo de partição com bom tamanho 38
Figura 15 – Ilustração de 3 partições sucessivas 39
Figura 16 – Imagem ilustrando como a posição da placa afeta a função 40
Figura 17 – Placa veicular sendo movida entre duas partições 41
Figura 18 – Valor da função em duas partições a medida que a placa se move entre elas . 41
Figura 19 – Valor da função a modelar, de ∞ a $+\infty$ 42
Figura 20 – Gráfico 3D da função à modelar 43
Figura 21 – Arquitetura Global da Implementação 45
Figura 22 – Erro de treinamento de redes neurais não-convolucionais 46
Figura 23 – Interface com usuário do *Marcador* 49
Figura 24 – Um *frame* de cada vídeo usado durante o desenvolvimento 51
Figura 25 – Interface gráfica do gerador de dados 52
Figura 26 – Região onde amostras são coletadas 53
Figura 27 – Exemplo de duas regiões iguais em *frames* diferentes 54
Figura 28 – Página do *TensorBoard* mostrando progresso do treinamento 56
Figura 29 – Grafo usado para treinamento da rede neural convolucional 56
Figura 30 – *Pipeline* de distorção de imagens 57
Figura 31 – *Pipeline* descrevendo a rede neural convolucional profunda 59
Figura 32 – Histograma temporal do erro de classificação 61
Figura 33 – Gráfico do erro de treinamento 61
Figura 34 – Interface com o usuário do módulo *Player* 62
Figura 35 – Contagem de acertos e erros de localização 64
Figura 36 – Gráfico da curva *Precision-Recall* (amostras de treinamento) 64
Figura 37 – Gráfico da curva *Precision-Recall* (amostras de teste) 66
Figura 38 – Gráfico 3D da função aplicada a carros 68
Figura 39 – Gráfico 3D da função aplicada a moto e ônibus 69
Figura 40 – Gráfico 3D mostrando a função aplicada à um *frame* inteiro 69
Figura 41 – Taxa de *frames* por segundo no módulo *Player* 70
Figura 42 – Tempo gasto em cada subtarefa no processamento de um *frame* 70

LISTA DE TABELAS

TABELA 1 – Arquivos de vídeo usados durante o desenvolvimento 50
TABELA 2 – Marcações feitas em cada vídeo 51
TABELA 3 – Exemplos de treinamento gerados para cada vídeo 55
TABELA 4 – Camadas da Rede Neural Implementada 60
TABELA 5 – Erros e acertos de localização 65
TABELA 6 – Erros e acertos de localização 67
TABELA 7 – Taxa de *Frames* por Segundo 67

LISTA DE SIGLAS

CNN	*Convolutional Neural Network*, rede neural convolucional
DCNN	*Deep Convolutional Neural Network*, rede neural convolucional profunda
ANN	*Artificial Neural Network*, rede neural artificial
ReLu	*Rectified Linear Unit*, unidade linear retificada
LBP	*Local binary patterns*, padrões binários locais
ORB	*Oriented FAST and rotated BRIEF*, FAST orientado e BRIEF rotacionado
HOG	*Histogram of oriented gradients*, histograma de gradientes orientados
SVM	*Support Vector Machine*, máquinas de vetores de suporte
RGB	*Red, green, blue*, vermelho, verde, vermelho
ILSVRC	*ImageNet Large Scale Visual Recognition Challenge*, desafio de reconhecimento de imagens em larga escala ImageNet
SIFT	*Scale-Invariant Feature Transform*, transformada invariante à escala
CIFAR	Canadian Institute for Advanced Research
OCR	*Optical Character Recognition*, reconhecimento ótico de caracteres
TDNN	*Time delay neural network*, rede neural de atraso de tempo
kNN	*k-Nearest Neighbours*, k vizinhos próximos
SSA	*Singular Spectrum Analysis*, análise de espectro singular
HWC	*Height X Width X Channel*, altura X largura X canal
HW	*Height X width*, altura X largura
JSON	*JavaScript Object Notation*, notação de objetos do *JavaScript*
AUR	*Arch User Repository*
CPU	*Central Processing Unit*, unidade central de processamento
BFS	*Breadth-first search*, busca em largura
GPU	*Graphical processing unit*, unidade gráfica de processamento

SUMÁRIO

1 INTRODUÇÃO 11
1.1 OBJETIVO 12
1.2 ESTRUTURA DA MONOGRAFIA 12
2 REDES NEURAIS CONVOLUCIONAIS 14
2.1 REDES NEURAIS NÃO-CONVOLUCIONAIS 15
2.2 REDES NEURAIS CONVOLUCIONAIS 17
2.2.1 Camadas convolucionais 19
2.2.2 Bordas 19
2.2.3 *Stride* 20
2.2.4 Profundidade do Filtro 21
2.2.5 Processamento em Lotes 21
2.2.6 Pooling 22
2.2.7 ReLu 22
2.2.8 Últimas Camadas 23
2.2.9 Rede Neural Convolucional Completa 23
2.2.10 Treinamento 24
2.3 USO EM PROCESSAMENTO DE IMAGENS 25
2.3.1 Redes Neurais Não-Convolucionais 26
2.3.2 Uso de descritores 26
2.3.3 Redes Neurais Convolucionais em Imagens 27
2.3.4 Uso de Cores 29
2.3.5 Reconhecimento de Objetos 29
2.3.6 Localização de Objetos 31
3 TRABALHOS RELACIONADOS 32
3.1 REDES NEURAIS CONVOLUCIONAIS APLICADAS A IMAGENS 32
3.2 APRENDIZADO DE MÁQUINAS PARA MONITORAMENTO DE TRÁFEGO 33
3.3 DETECÇÃO DE PLACAS DE VEÍCULOS 34
4 LOCALIZAÇÃO DE PLACAS VEICULARES USANDO REDES NEURAIS CONVOLUCIONAIS 36
4.1 TAMANHO DAS PARTIÇÕES 37
4.2 *STRIDES* DAS PARTIÇÕES 38
4.3 FUNÇÃO A SER MODELADA 39
4.4 ARQUITETURA DA REDE NEURAL 43
5 IMPLEMENTAÇÃO E EXPERIMENTOS 44
5.1 ARQUITETURA GLOBAL 44
5.2 ABORDAGENS ANTERIORES 45
5.3 ESCOLHA DE TECNOLOGIAS 47
5.4 RECURSOS DE HARDWARE 48
5.5 IMPLEMENTAÇÃO DOS MÓDULOS DE *SOFTWARE* 49
5.5.1 Marcador 49
5.5.2 Gerador de Dados de Treinamento 51

5.5.3 Treinador 55
5.5.4 Player 62
5.6 EXPERIMENTOS E DESEMPENHO 63
5.6.1 Desempenho de Localização 63
5.6.2 Função 66
5.6.3 Tempo de Classificação 67
6 CONCLUSÕES 71
REFERÊNCIAS 72

1 INTRODUÇÃO

Figura 1: Resultado esperado (autoria própria).

Soluções de monitoração e fiscalização de veículos apresentam uma grande demanda por sistemas de visão computacional. Seja para contar tráfego, fiscalizar o uso das vias, monitorar rodízios ou para cobrar pedágios, a capacidade de obter informações a partir de imagens é de grande interesse, sendo parte integrante de uma solução típica (ANAGNOSTOPOULOS et al., 2008). Existem soluções propostas para parte destes problemas, no entanto há a possibilidade da busca por estratégias com menor custo ou melhor desempenho, à medida que avanços no campo do reconhecimento de imagens acontecem.

Redes neurais convolucionais (CNN, convolutional neural networks) são tipos de redes neurais biologicamente inspiradas no conceito de campos receptivos (HUBEL; WIESEL, 1968). Este tipo de rede neural pode ser alimentado diretamente com os *pixels* da imagem (LECUN et al., 1998). O advento das Redes Neurais Convolucionais Profundas (DCNN, deep CNN) viabilizou um grande salto no desempenho de classificação para alguns casos. O uso de mais camadas permite níveis maiores de abstração na análise da imagem, o que resulta em maior taxa

de acerto, ao custo de maior tempo de treinamento. Este tipo de rede neural representa o atual estado-da-arte em reconhecimento de imagens (SZEGEDY et al., 2015a).

Este projeto demonstra uma abordagem para localização aproximada de placas veiculares mediante à aplicação direta de DCNN aos pixels de partições dos *frames* de um vídeo. O sistema de particionamento da imagem evita a aplicação da rede neural usando janela deslizante *pixel-a-pixel*, permitindo que a localização das placas veiculares em um vídeo de 480×768 a uma taxa de 14,8 FPS, ou 67,6 ms por *frame* rodando em um notebook Core i5 com GPU GTX 750M.

O projeto demonstra um método que permite o treinamento da rede neural de forma que ela seja robusta contra alguns fatores, como ruído e iluminação, mesmo quando esta rede neural é treinada sem estes componentes. Isso permite que a rede neural seja aplicada aos *frames* sem necessidade de pré-processamento, como filtragem ou normalização.

O método proposto usa a rede neural para fazer a regressão de uma função escalar aplicada a uma imagem, de forma que o resultado da função permite determinar se uma placa está na imagem. Uma função específica é parte do método que será proposto.

Será apresentado o desempenho de localização de placas usando medidas padronizadas, especificamente *precision* e *recall*, juntamente com a curva PR (POWERS, 2011). Também será analisada a capacidade da rede neural de reproduzir a função para a qual foi treinada. Finalmente será medido o desempenho em *frames* por segundo do sistema quando aplicado alguns vídeos diferentes, e quando executando com e sem GPU.

1.1 OBJETIVO

O objetivo deste trabalho é desenvolver uma abordagem para treinamento e uso de redes neurais convolucionais para para a construção de um sistema robusto e eficiente de localização de placas de veículos em vídeo, e realizar uma implementação para medição de dados de desempenho. A figura 1 mostra um exemplo de saída que se pretende obter.

1.2 ESTRUTURA DA MONOGRAFIA

O restante deste trabalho é organizado da seguinte forma: O capítulo 2 discorre sobre redes neurais convolucionais em geral, comparando-as com as redes neurais não-convolucionais. Como este tópico é relativamente novo optou-se por fazer sua apresentação o mais cedo possível no documento. O capítulo 3 é uma revisão da bibliografia atual sobre detecção e localização de

placas veiculares. O capítulo 4 apresenta uma proposta de um método para aplicar redes neurais convolucionais para localizar placas veiculares. O capítulo 5 apresenta uma implementação do método proposto, experimentos e os seus resultados. O capítulo 6 apresenta as conclusões deste trabalho.

2 REDES NEURAIS CONVOLUCIONAIS

Rede neural artificial (ANN, do inglês, *Artificial Neural Network*) é um modelo computacional inspirado na forma com que o cérebro resolve problemas (GILBERT; TERNA, 2000). Este modelo possui unidades, denominadas neurônios, que possuem um valor que é calculado como uma função do valor de outros neurônios.

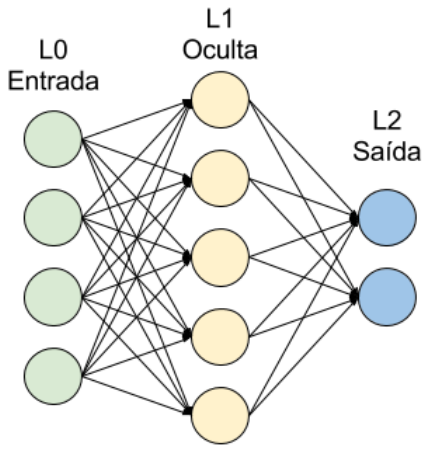

Figura 2: Exemplo de rede neural totalmente conectada (autoria própria).

Alguns neurônios especiais, denominados “entradas” possuem um valor obtido de fora do sistema. Estes são especiais porque são os únicos cujos valores não são calculados. O valor de alguns dos neurônios de uma rede neural artificial podem afetar sistemas externos a ela. Por isso são designados “saídas”. Neurônios que não são de entrada ou de saída são chamados “ocultos”. A figura 2 ilustra uma rede neural simples com os três tipos de neurônios.

Neste capítulo será apresentada parte da teoria de redes neurais convolucionais e não-convolucionais. Na seção 2.1 apresentam-se as redes neurais não-convolucionais, explorando algumas das suas propriedades matemáticas. Na seção 2.2 será feita uma apresentação das redes convolucionais, explorando vários de seus aspectos matemáticos e práticos.

2.1 REDES NEURAIS NÃO-CONVOLUCIONAIS

As redes neurais artificiais são tipicamente organizadas em "camadas", e a escolha dos tipos de neurônio, juntamente com a forma com que os neurônios são conectados, é denominada "topologia". As redes são classificadas como *feedforward* (ou diretas) quando as conexões não formam ciclos ou *recurrent* (recorrentes) quando formam. Se uma camada está conectada a todos os neurônios da camada anterior diz-se que a camada é totalmente conectada. A definição da função de transferência do neurônio é uma parte importante da topologia das redes neurais. Um tipo muito comum de neurônio é definido por:

$$v = A\left(\left(\sum_n w_n i_n\right) + b\right) \tag{1}$$

Onde v é o valor da saída do neurônio, i_n é n-ésima entrada do neurônio, w_n é um escalar associado a esta entrada, denominado peso, e b é um escalar arbitrário, denominado *bias*. Este último valor permite que um neurônio produza valores não-nulos mesmo que a entrada seja nula. A é uma função denominada função de ativação do neurônio. Essa função pode ser usada para tornar o neurônio não-linear, como no caso da tangente hiperbólica e da função ReLu, ou linear retificada (ver figura 3). Os pesos w_n determinam a influência de cada entrada do neurônio.

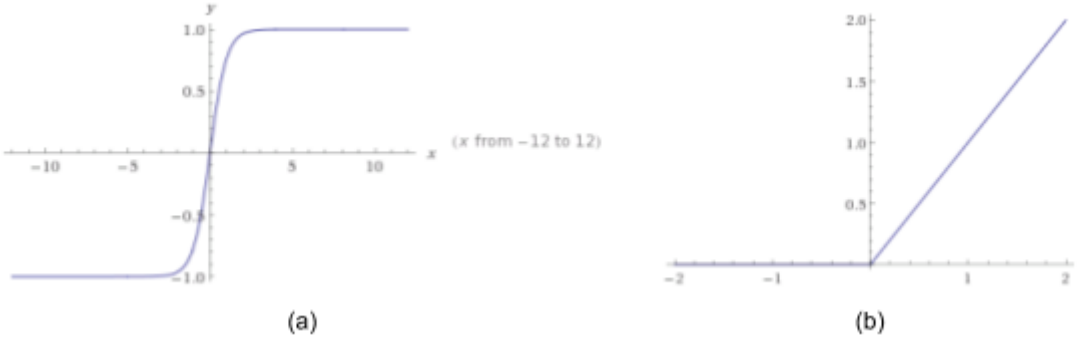

Figura 3: Comparação entre a função tangente hiperbólica e a Relu. Em (a) está a função tangente hiperbólica e em (b) está a função ReLu (autoria própria).

A descrição de redes neurais com este tipo de neurônio e com topologia totalmente conectada é especialmente conveniente. As três camadas da rede neural ilustrada na figura 2 podem ser representadas por matrizes:

$$L0_{4\times1} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \quad (2) \qquad L1_{5\times1} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{pmatrix} \quad (3) \qquad L2_{2\times1} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \quad (4)$$

Para calcular a primeira camada intermediária é preciso aplicar a equação (1) cinco vezes, uma para cada neurônio. Porém, se esta camada for totalmente conectada, e todos os neurônios usarem a mesma função de transferência A, e a representação em matrizes estiver sendo usada, o cálculo de todos os neurônios desta camada pode ser realizado usando:

$$L1_{5\times1} = A_V\left(W1_{5\times4} \times L0_{4\times1} + B1_{5\cdot1}\right) \quad (5)$$

Onde A_V é uma versão vetorial da função de ativação resultante da aplicação da função A em cada um dos elementos do vetor. Os pesos dos cinco neurônios, denominados w_n na equação 1 são aqui representados por uma única matriz $W1_{5\times4}$, e os *bias* dos cinco neurônios, b, são representados por uma única matriz $B1_{5\times1}$. Da mesma forma, a camada de saída pode ser calculada com:

$$L2_{2\times1} = A_V\left(W2_{2\times5} \cdot L1_{5\times1} + B2_{2\times1}\right) \quad (6)$$

As matrizes W e B são os parâmetros que precisam ser aprendidos durante o treinamento, e são denominados parâmetros treináveis. O número de parâmetros de uma uma rede neural é igual ao número de valores contidos em todas as matrizes de pesos e *bias*. Quanto maior o número de parâmetros, mais flexibilidade a rede neural possui, porém mais lento é o treinamento. Se o número de parâmetros for excessivo a rede neural perde a capacidade de generalizar e ocorre *overfitting*.

De acordo com (HAWKINS, 2004), *overfitting* ocorre quando um modelo é mais flexível do que precisa ser. Quando um modelo destes é treinado, especialmente com poucos exemplos, o modelo pode acabar "memorizando" os exemplos, ao invés de generalizar o comportamento geral dos dados, fazendo com que o resultado do treinamento seja bom quando medido com o conjunto de treinamento, mas haja erro excessivo quando o mesmo modelo é aplicado em dados novos.

Como as dimensões da matriz de pesos de uma camada totalmente conectada são $d_{en} \times d_{sai}$, onde d_{en} é o número de neurônios na camada de entrada, e d_{sai} é o número de neurônios

da própria camada, então o número de parâmetros nessa matriz é igual ao produto dos dois. Se esta camada tiver 1000 entradas e 1000 saídas a matriz de pesos vai possuir 1.000.000 de parâmetros e a matriz de *bias* mais 1.000, resultando em um total de 1.001.000.

2.2 REDES NEURAIS CONVOLUCIONAIS

Redes neurais convolucionais são um tipo de rede neural que inclui operações baseadas em uma definição relaxada de convolução. A principal operação neste tipo de rede neural é a a correlação cruzada (*cross-correlation*). Para duas funções discretas f e g, a correlação cruzada discreta entre elas é definida por:

$$def : (f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m]g[m+n] \tag{7}$$

Onde f^* é o complexo conjugado da função f. A figura 4 ilustra uma operação de correlação cruzada sendo realizada representada graficamente.

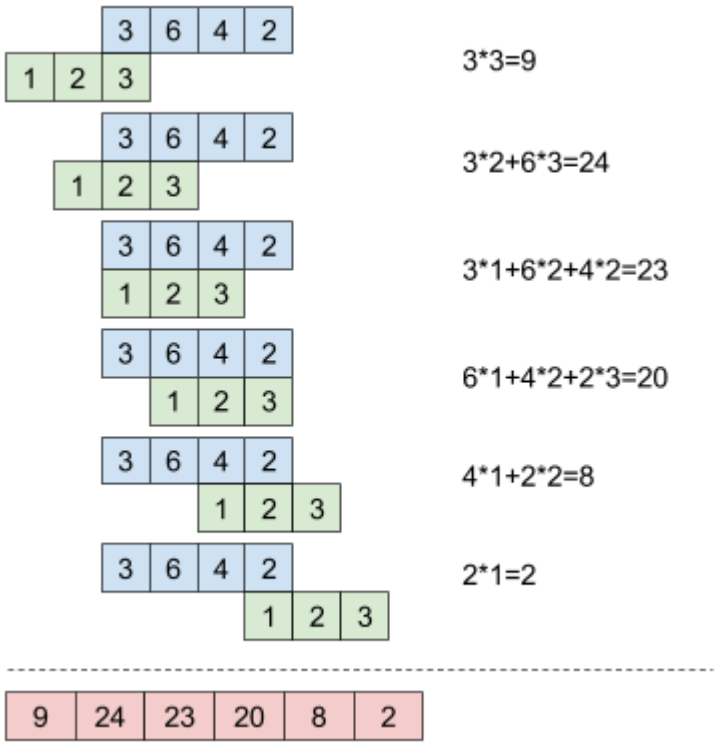

Figura 4: Exemplo de correlação cruzada 1D. Na imagem está sendo calculada a correlação cruzada entre $[3,6,4,2]$ e $[1,2,3]$. O primeiro conjunto de dados fica fixo enquanto o segundo vai sendo deslocado, e após cada passo se calcula a soma do produto, conforme ilustrado. O resultando é $[9,24,23,20,8,2]$. Todos os vetores representados possuem uma quantidade infinita de zeros à direita e à esquerda (autoria própria).

A correlação cruzada também é definida para dimensões maiores que 1. Tomando as funções discretas $f[n_1, n_2]$ e $g[n_1, n_2]$ pode-se escrever a convolução 2D destas funções como sendo:

$$def : (f \star\star g)[n_1, n_2] = \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} f^*[m_1, m_2] g[m_1 + n_1, m_2 + n_2] \tag{8}$$

É possível realizar convoluções para um número arbitrário de dimensões através da generalização desta equação.

A aplicação do operador de correlação cruzado é próximo ao conceito de filtro linear, bastante usado em processamento digital de imagens (GONZALEZ; WOODS, 2007). Os conceitos não são idênticos porque filtro digital trata uma das entradas como sendo "sinal", e a outra como sendo o filtro, ou *kernel*, e convoluções e correlações cruzadas tratam as duas entradas de forma idêntica, e produzem um resultado diferente nos extremos. Como observa-se na figura 4, a saída é um vetor de dimensão 6, e um filtro digital geraria saída de tamanho 4 (se houvesse extensão de borda). No entanto, os números obtidos pelo filtro linear são iguais aos obtidos pela correlação cruzada nas posições centrais. Mais detalhes sobre isso serão cobertos na seção 2.2.2.

A correlação cruzada entre $f[n]$ e $g[n]$ é igual à convolução entre $f[n]$ e $g[-n]$, e relações semelhantes existem para qualquer dimensionalidade. Uma camada de uma rede neural que use qualquer uma das operações é denominada "convolucional", pois neste contexto as operações são intercambiáveis. Ao se trocar convolução por correlação cruzada os mesmos parâmetros serão aprendidos em uma ordem diferente. Por este motivo os termos "convolução" e "convolucional" são usados de forma relaxada no contexto de redes neurais. Conforme será visto adiante, existe mais um motivo para isso.

O uso de convoluções em redes neurais, especialmente para dimensões superiores a 1, requerem que os dados nos quais este operador vai ser executado sejam representados de forma a preservar a geometria original. Se os dados forem uma imagem bidimensional, por exemplo, essa característica precisa ser preservada, e a rede neural precisa conhecer a forma dos dados.

Uma das formas de estruturar os dados é usando tensores, que são uma extensão de vetores que admite um número arbitrário de dimensões. O motivo para isso é permitir preservar a geometria da informação que está sendo processada.

Se a rede neural convolucional for aplicada em uma imagem bidimensional com di-

mensões H e W com C canais de cor, ela pode ser representada por um tensor de dimensões $H \times W \times C$. Uma imagem tridimensional com profundidade D é representada por um tensor $D \times H \times W \times C$, e assim por diante. Isso permite preservar o posicionamento relativo das informações e viabiliza aplicar o operador de correlação cruzada.

Caso a rede neural esteja operando sobre um tipo de informação que não possui o conceito de "canal", como séries temporais, é conveniente artificialmente criá-lo. Esse é o caso de séries temporais com "N" entradas, que serão representadas por um tensor com dimensões $N \times 1$. O motivo para isso é uniformidade, como será explicado adiante.

2.2.1 CAMADAS CONVOLUCIONAIS

Uma camada convolucional é definida pela a aplicação de um *kernel* sobre a sua entrada usando o operador convolução. Como já foi mencionado, o uso do termo convolução é bastante relaxado neste caso por que, além de normalmente referir-se a uma correlação cruzada, é aplicado de forma idêntica à aplicação de um filtro digital, gerando as diferenças já discutidas no resultado desta operação.

O filtro da camada convolucional é um tensor que contém parâmetros treináveis. Se a entrada da camada convolucional tem dimensões $D0 \times D1 \times D2 \times ... \times C$, onde C é o número de canais, o filtro terá o mesmo número de dimensões, sendo que o tamanho de todas as dimensões, exceto a última são hiperparâmetros arbitrários. A última dimensão do filtro precisa ser igual ao número de canais da imagem de entrada da camada convolucional, como ilustrado na figura 5.

Camadas convolucionais também usam *bias*, porém em uma quantidade muito menor que as redes neurais totalmente conectadas. Se uma camada totalmente conectada possui 100 neurônio ela vai ter 100 *bias*. No caso de uma camada convolucional, porém usa-se um único *bias* por convolução. Na figura 5 existe apenas um bias, com o qual todos os valores da saída serão somados.

2.2.2 BORDAS

Como já foi mencionado, a convolução é aplicada de forma idêntica a um filtro linear (GONZALEZ; WOODS, 2007), e assim como no caso dos filtros lineares, existem algumas opções para tratamento de bordas dos dados.

Na figura 5 o tamanho do tensor de saída foi reduzido de 6 para 4. Neste caso a opção foi por não estender as bordas. Por este motivo, o resultado tem que ser reduzido para impedir

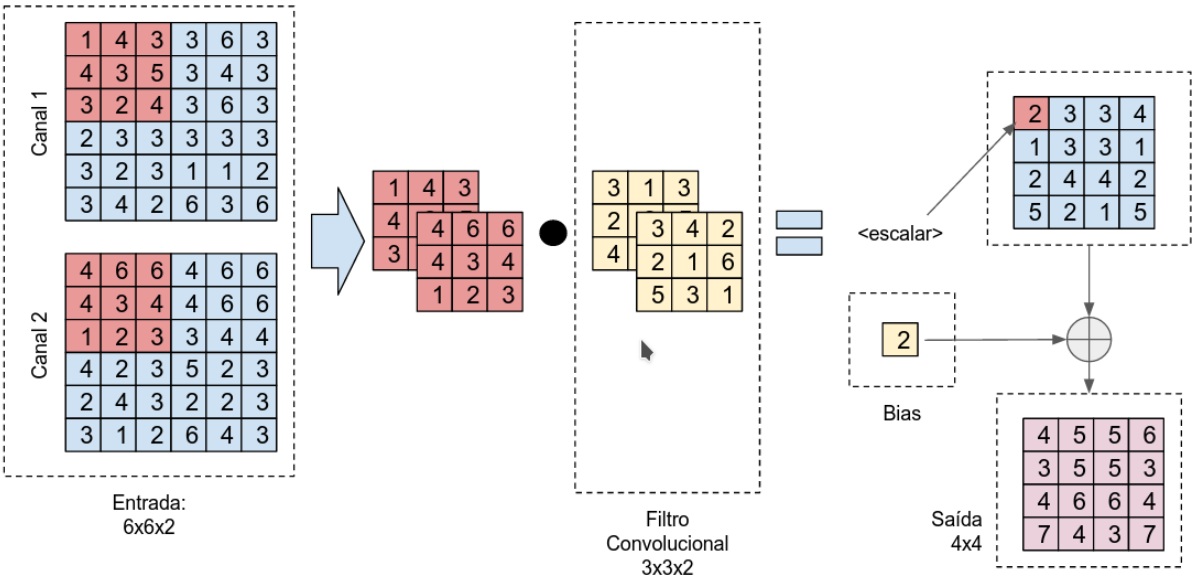

Figura 5: Exemplo de convolução 2D. Exemplo da aplicação de uma única convolução a uma imagem de entrada $6 \times 6 \times 2$. O tamanho da última dimensão do filtro e da entrada precisam ser iguais, no caso 2. As dimensões 3×3 do filtro foram escolhidas arbitrariamente. O símbolo do círculo na imagem representa um produto interno. A imagem de saída foi gerada deslocando a seleção de *pixel* a *pixel* até cobrir toda a imagem de entrada. Como as bordas não foram estendidas, a imagem resultante é menor que a de entrada (autoria própria).

que o filtro possua termos despareados.

Pode ser desejável fazer a saída ter o mesmo tamanho da entrada. Para isso é necessário estender as bordas da entrada, o que permitiria ao filtro ser deslocado por toda a extensão entrada. Em camadas convolucionais observou-se apenas a extensão com zeros. O software que foi adotado para a implementação deste trabalho não suporta extensão que não seja com zeros, e não foi encontrado artigo usando método diferente, como repetição.

A opção de extensão de borda é um hiperparâmetro.

2.2.3 *STRIDE*

Ao se aplicar o filtro de convolução no tensor de entrada pode-se movê-lo de posição a posição, até cobrir todos os locais válidos em cada direção, como ilustrado na figura 5, ou pode-se desejar aplicar a cada "n_0" *pixels* em uma direção, "n_1" *pixels* em outra direção, e assim por diante. Esta opção é denominada *stride*.

Quando uma camada convolucional possui como entrada um tensor de dimensões $d1 \times d2 \times ... \times dj \times C$, o *stride* é definido como sendo um tensor unidimensional de tamanho j. O *stride* $[1,2,1]$, por exemplo, indica que na primeira dimensão o filtro será aplicado em todas as posições válidas, na segunda dimensão será aplicado a cada 2 *pixels* e na terceira será aplicada

novamente em todas as posições válidas. *Stride* maior que 1 causa redução no tamanho do tensor de saída.

Se um *stride* 2×2 fosse aplicado na convolução da figura 5 a imagem de saída seria 2×2.

2.2.4 PROFUNDIDADE DO FILTRO

A figura 5 mostrou um único filtro sendo aplicado ao tensor de entrada. No entanto, é possível aplicar um número arbitrário deles, sendo que cada filtro produz um tensor de saída. O número de filtros é denominado "profundidade" da camada convolucional.

A camada convolucional vai produzir uma saída para cada filtro. É conveniente agrupar todas as saídas em um único tensor. Se a entrada da convolução possui dimensões $d0 \times d1 \times ... \times C$ e a profundidade da camada convolucional é D pode-se representar as D saídas com um tensor de dimensões $d0 \times d1 \times ... \times D$. Observar que o tensor de saída agora é semelhante ao tensor de entrada, substituindo apenas C no tensor de entrada por D no tensor de saída. Se houver uma camada convolucional depois desta ela vai tratar a última dimensão da mesma forma que esta trata o canal, o que é conveniente. Se na figura 5 fossem aplicados 8 filtros a saída seria um tensor $4 \times 4 \times 8$.

O tensor que representa o filtro também precisa receber uma dimensão adicional, para poder representar todos os D filtros, e também vai ser a última dimensão do tensor. Sendo assim, para aplicar em uma imagem RGB 8 filtros 5×5 o tensor que representa os pesos das convoluções possui dimensões $5 \times 5 \times 3 \times 8$.

2.2.5 PROCESSAMENTO EM LOTES

Durante o treinamento das redes convolucionais o otimizador vai fazer alterações nos valores dos parâmetros treináveis e precisa determinar se elas melhoraram ou pioraram o desempenho. No caso de treinamento supervisionado a rede neural é alimentada com dados e o resultado da classificação é comparado com o valor esperado. É conveniente usar múltiplas imagens para este teste, para que ele seja o mais representativo possível.

O número de imagens fornecidas é denominado "tamanho do lote". Se uma rede neural trabalha com dados de dimensões $d0 \times d1 \times ... \times C$ podem-se agrupar B exemplos em um único tensor, com dimensões $B \times d0 \times d1 \times ... \times C$. As camadas convolucionais tratam cada uma das imagens separadamente, conforme já foi descrito, e emitem na sua saída um único tensor com dimensões $B \times d0 \times d1 \times ... \times D$, sendo D a profundidade da camada. A figura 6 ilustra as

entradas e saídas de uma camada convolucional usando lotes e profundidade.

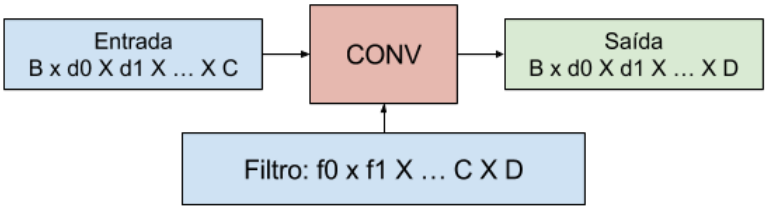

Figura 6: Convolução com múltiplos filtros e entrada em lote. Diagrama de uma camada convolucional de profundidade *D* sendo aplicada em um lote contendo *B* imagens. Ilustração não inclui *bias* (autoria própria).

2.2.6 POOLING

Após a aplicação de uma camada convolucional pode-se aplicar uma camada de *pooling*, que é uma forma de subamostragem. A figura 7 ilustra um dos tipos de *pooling*, denominado *maxpool*.

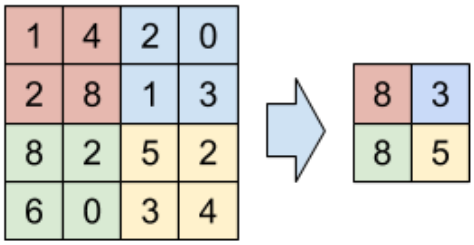

Figura 7: Exemplo de *maxpool* 2×2 aplicado usando *stride* 2×2. O filtro move de dois em dois *pixels* e possui tamanho 2 em cada dimensão (autoria própria).

Esta operação possui como parâmetros o tamanho do filtro, o *stride* e a opção de borda. A operação a ser aplicada no filtro pode ser *max* ou *avg* (média), que definem respectivamente os filtros *maxpool* e *avgpool*.

Como as operações de *pooling* são normalmente feitas com *stride* maior que 1 elas acabam reduzindo consideravelmente o tamanho do tensor de saída. No caso de um *maxpool* 2×2, por exemplo, o tensor resultante vai ter 25% do número de valores do tensor de entrada.

Uma camada convolucional com *pooling* pode ser contada como uma única camada ou como duas, dependendo do caso. Em algumas arquiteturas mais complexas, como em (SZEGEDY et al., 2015a), onde definem-se camadas do tipo *inception*, pode não ser possível associar uma operação de *pooling* a uma única convolução, requerendo contagem separada.

2.2.7 RELU

Assim como em redes neurais totalmente conectadas, pode-se aplicar uma função de ativação para tornar a camada não-linear. As funções tangente hiperbólica, sigmoide ou outras tipicamente usadas em redes neurais não-convolucionais podem ser aplicadas. No entanto a função ReLu, ou linear retificada, resulta em treinamento substancialmente mais rápido enquanto mantém a capacidade de generalização da rede neural treinada. A função ReLu é definida por:

$$ReLu(x) = max(0,x) \tag{9}$$

Quando uma camada de uma rede neural inclui apenas elementos lineares, ela é dita linear. O operador de convolução em si é linear, então isso vai ocorrer quando a camada não inclui uma função de ativação, como a ReLu e não possui outros elementos não-lineares, como *maxpool*, caso o *pooling* esteja sendo considerado como parte da camada convolucional.

2.2.8 ÚLTIMAS CAMADAS

Após as camadas convolucionais terem sido aplicadas é necessário usar um classificador, mecanismo de regressão ou outro sistema que gere o tipo de saída desejada para a rede neural. Uma das possíveis formas de realizar esta função é usar uma ou duas camadas totalmente conectadas, como ilustrado na figura 8. No exemplo a função de ativação da penúltima camada é ReLu e a última camada é linear.

2.2.9 REDE NEURAL CONVOLUCIONAL COMPLETA

Para alguns casos simples, pode-se construir uma rede neural convolucional conectando-se uma camada convolucional a uma *maxpool* e uma ReLu. Este conjunto pode ser repetido algumas vezes até que o número de saídas da camada convolucional seja baixo o suficiente para que seja passado por duas camadas totalmente conectadas. Se houver interesse em não reduzir o tamanho total do tensor pode-se omitir a camada *maxpool*. Um exemplo dessa topologia é mostrado na a figura 8.

A primeira camada da rede neural vai ter filtros que serão sensíveis a características simples da entrada, que no caso de imagens seriam, por exemplo, gradientes e trechos de linhas. Em cada camada seguinte estas características são recombinadas com as detectadas nas regiões vizinhas, formando conceitos progressivamente mais sofisticados a respeito da imagem.

A topologia da rede neural e o conjunto de todos os hiperparâmetros definem o número de parâmetros treináveis que a rede neural vai possuir e quantas operações são necessárias para aplicar a rede neural em um (ou um lote de) amostras. Uma rede neural mais larga, com um maior número de filtros por camadas, possui a capacidade de aprender mais características. Redes neurais mais profundas possuem capacidade de abstração maior, sendo capazes de inferir a partir dos dados de entrada conceitos mais complexos. Se a rede neural for mais complexa do que o necessário pode haver *overfitting*.

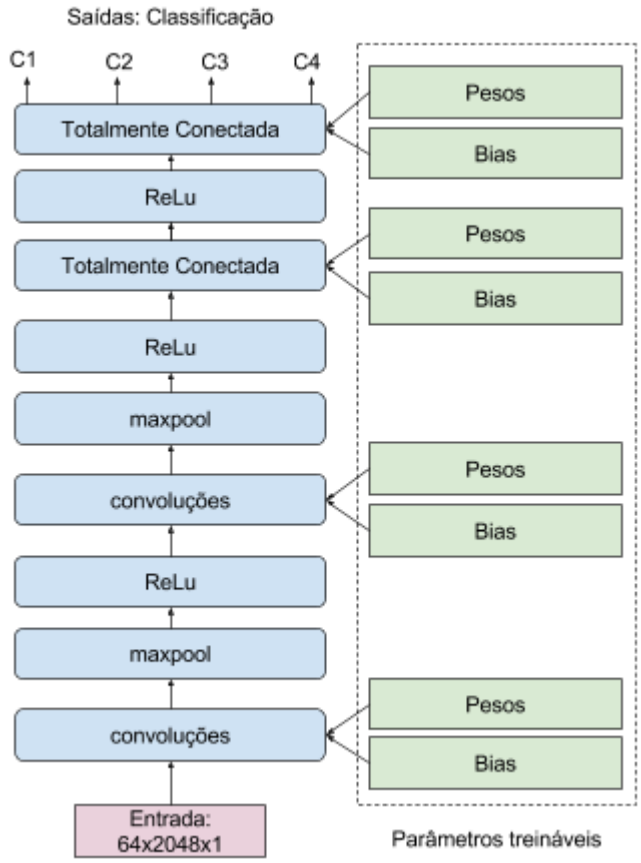

Figura 8: Exemplo de rede neural convolucional completa. Esta rede neural possui nove camadas para classificação de séries temporais em uma de 4 classes. Destacam-se os parâmetros treináveis (autoria própria).

2.2.10 TREINAMENTO

O treinamento de redes neurais convolucionais é feito de forma idêntica ao treinamento de redes não-convolucionais. Para o caso de treinamento supervisionado é necessário que se tenha acesso de exemplos rotulados. Cada exemplo é formado por duas partes: entrada e saída esperada. A entrada é denominada x, e é o conjunto de todos os valores que serão fornecidos para a rede neural. A saída esperada, denominada y, é o que se espera obter da rede neural.

Quando a entrada x é fornecida para a rede neural ela vai produzir uma saída, denominada saída estimada, ou $\widehat{y}$.

A rede neural possui parâmetros denominados hiperparâmetros e parâmetros treináveis. O primeiro define características que são escolhidas manualmente por quem está fazendo o treinamento. Isso envolve, entre outras coisas, o número de camadas, e a quantidade de camadas. Pode-se dizer que a própria escolha do método de classificação, como CNN ou HOG+SVM é um hiperparâmetro. Os parâmetros treináveis são aqueles que são encontrados pelo processo de treinamento.

O treinamento ocorre guiado por um valor denominado perda da rede neural, ou *loss*. A perda é um escalar real que mede o desempenho da rede neural. Não importa quantas saídas a rede neural possui, ela vai sempre ter um único real que representa a perda. Existem várias formas de se calcular este valor, dependendo do uso da rede neural. Quando a rede neural faz regressão é comum o uso da função de perda L2:

$$\text{L2 loss} = \frac{1}{N}\sqrt{\sum_{n=1}^{N}(y_n - \widehat{y}_n)^2} \tag{10}$$

Onde a raiz quadrada normalmente não é calculada, e algumas vezes usa-se a soma ao invés da média. Também usa-se a perda L1, que é a média do módulo da diferença entre os valores estimados e os esperados, e para classificação usa-se a perda logística, ou entropia cruzada (ROBERT, 2014).

O treinamento é realizado por um "otimizador", cujo papel é atualizar os parâmetros treináveis para minimizar a perda. Os otimizadores mais usados são derivados do *gradient descent*, sendo Kingma e Ba (2014) um dos melhores disponíveis.

Como otimizador depende da função de perda para atualizar os parâmetros da rede neural, este valor precisa o mais representativo possível. Por isso ele deve ser calculado em função de vários exemplos, não apenas um.

Durante o processo de treinamento um conjunto de exemplos vai ser usado e os pesos vão ser atualizados uma vez, então um conjunto diferente de dados vai ser usado para a interação seguinte. É importante que os exemplos estejam bem embaralhados e contenham quantidades próximas de exemplos dos diversos tipos que a rede neural vai processar. Se a rede neural estiver classificando imagens em dois grupos $A1$ e $A2$, por exemplo, e todos os exemplos de uma classe vierem depois da outra o treinamento não vai funcionar. Idealmente, a cada interação do otimizador deve haver exemplos dos dois tipos.

2.3 USO EM PROCESSAMENTO DE IMAGENS

O uso de redes neurais convolucionais é uma alternativa a diversos métodos já existentes de detecção e reconhecimento de objetos em imagens. Aqui serão mostradas algumas das alternativas usuais, para que sejam contrastadas com a classificação baseada em redes neurais convolucionais.

2.3.1 REDES NEURAIS NÃO-CONVOLUCIONAIS

Quando uma imagem vai ser processada por uma rede não-convolucional ela precisa ser convertida para a forma plana, em um vetor com dimensões $H \cdot W \cdot C \times 1$.

O uso de camadas totalmente conectadas é proibitivo para classificação de imagens desta maneira. Se uma rede neural for usada para processar imagens de 1 *megapixel*, ou 720×1280, e a primeira camada possuir um número de neurônios igual ao número de neurônios da camada de entrada, a matriz de pesos teria $(720 \cdot 1280 \cdot 3)^2 \approx 7.6 \cdot 10^{12}$ parâmetros. Se for representado por números ponto-flutuante de 32 bits isso ocuparia mais de 30 TiB. O mesmo processo em uma imagem 480px por 640px geraria quase 850 bilhões de parâmetros só na matriz de pesos.

Outras topologias mais esparsas podem ser construídas, mas este tipo de rede neural possui outros problemas que as tornam inviáveis para processamento direto de imagens. Um exemplo é o fato delas não serem invariantes ao deslocamento. Se a rede neural aprende a reconhecer um *feature* em um local da imagem, não vai conseguir reaproveitar essa capacidade em outras posições. Portanto teria que aprender a mesma *feature* em cada possível posição onde ela possa aparecer, e fazer isso para todas as *features*.

Como a primeira operação feita foi converter a imagem para um formato "plano", a geometria da imagem foi destruída. Não é mais possível determinar quando dois *pixels* estão próximos, e essa é uma informação muito importante sobre a imagem.

2.3.2 USO DE DESCRITORES

Uma das abordagens possíveis para reconhecimento e detecção de imagens é o uso de descritores. Descritores são operações que tomam uma imagem como entrada e resumem as sua características em um conjunto menor de informações. Exemplos de descritores muito usados são LBP (WANG; HE, 1990), ORB (RUBLEE et al., 2011), HOG (DALAL; TRIGGS, 2005), Haar-Wavelets (NABOUT; TIBKEN, 2008), filtros de Gabor (RIAZ et al., 2012). O treinamento

é realizado sobre as *features* extraídas pelo descritor escolhido usando um classificador como redes neurais ou SVM (BOSER et al., 1992) (CORTES; VAPNIK, 1995).

Para usar esta abordagem um descritor precisa ser escolhido e configurado, muitas vezes manualmente. O desempenho do sistema como um todo vai depender dessas escolhas. Se a operação de detecção depender de conceitos complexos, envolvendo correlação entre múltiplas características, o descritor e o classificador precisam ser escolhidos especificamente para isso.

2.3.3 REDES NEURAIS CONVOLUCIONAIS EM IMAGENS

As redes neurais convolucionais operam diretamente nos *pixels* da imagem, não sendo necessário escolher um descritor, e não possuem os problemas que as redes não-convolucionais possuem. As camadas convolucionais se comportam como descritores, porém os parâmetros são aprendidos como parte do processo de treinamento, portanto são otimizados para responderem precisamente para os exemplos que forem fornecidos. Apenas alguns hiperparâmetros, como tamanho da convolução, número de filtros e de camadas precisam ser escolhidos mediante decisões de meta de precisão e *recall*, tempo para classificação e complexidade dos conceitos a serem aprendidos.

Para que seja possível usar convoluções é preciso preservar a geometria dos dados. Se uma imagem de 40 *pixels* de altura por 120 *pixels* de largura contendo 3 canais de cor for usado na entrada de uma rede neural convolucional, ela precisa ser representada por um tensor apropriado, como $32 \times 40 \times 120 \times 3$. Este tensor permite alimentar a rede neural com um lote de 32 imagens.

Para aplicar uma camada convolucional à imagem de entrada escolhe-se o tamanho da convolução, o número de filtros e o modo de operação nas bordas. A figura 9 ilustra um filtro 3×3 sendo aplicado a uma imagem RGB. Para que o filtro tenha profundidade 64, por exemplo, o tensor que define o filtro será $3 \times 3 \times 3 \times 64$ e o tensor resultante da convolução será $32 \times 40 \times 120 \times 64$, considerando que seja usado preenchimento nas bordas.

Para finalizar a camada convolucional adiciona-se um *bias*. Para isso usa-se um escalar para cada uma das 64 imagens resultantes. O tensor que define os *bias* é um tensor unidimensional com tamanho 64.

Camadas *maxpool* e ReLu podem ser adicionadas para reduzir as dimensões da imagem ou aumentar a sua não-linearidade. A figura 10 ilustra o uso das três camadas em sequência.

A imagem é tratada por camadas sucessivas de convolução, *maxpool* e ReLu até que o tamanho total do tensor seja pequeno o suficiente para poder ser processado por duas camadas

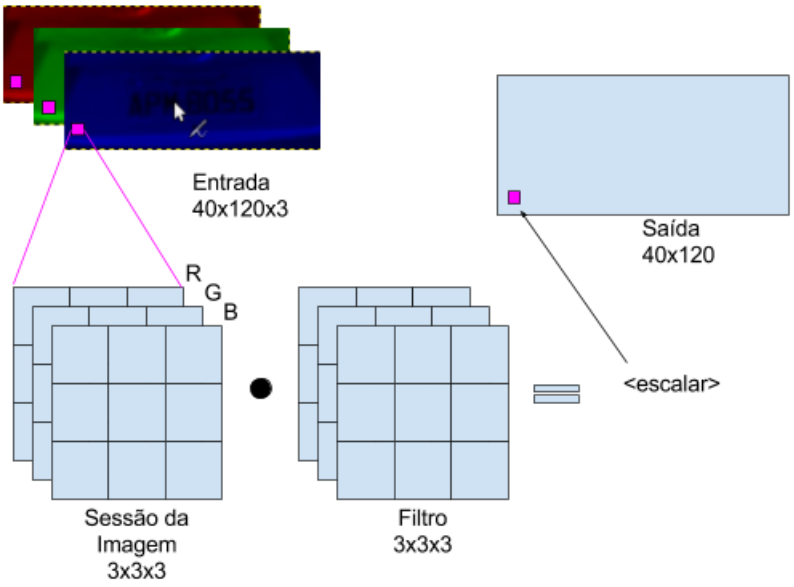

Figura 9: Convolução sendo aplicada em uma imagem RGB. Como a imagem possui 3 canais o filtro será $3 \times 3 \times 3$. Uma partição da imagem original com o mesmo tamanho do filtro, $3 \times 3 \times 3$ é extraída da imagem, e um produto interno é realizado entre os dois, resultando em um escalar. Este escalar é armazenado no tensor de saída nas coordenadas corretas. A ilustração só mostra uma imagem de entrada e só um filtro e, portanto, uma imagem de saída (autoria própria).

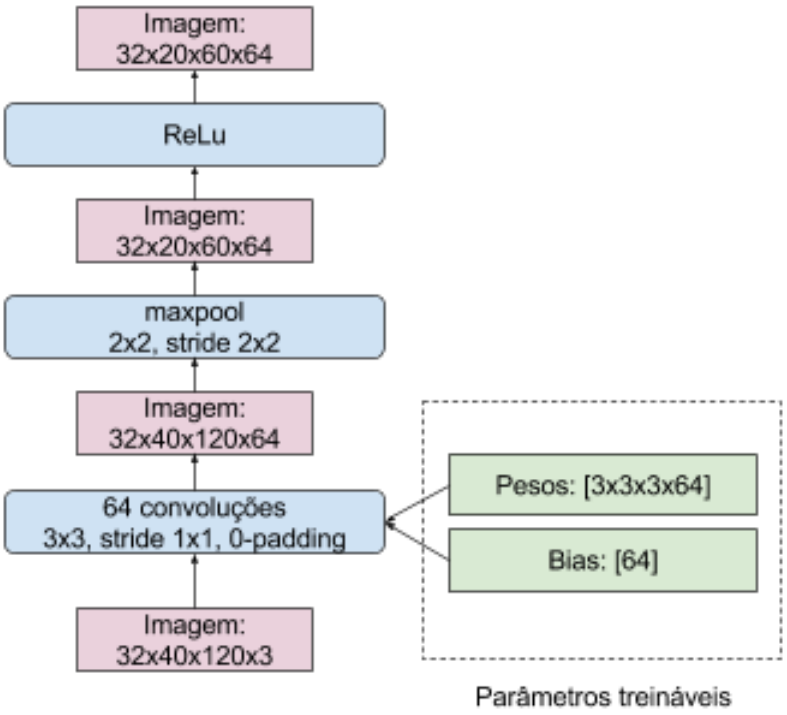

Figura 10: Exemplo de rede neural convolucional completa aplicada à uma imagem RGB. Uma sequência de operações é realizada usando uma rede neural convolucional incluindo 64 convoluções 3×3, *maxpool* e ReLu aplicadas em um lote de 32 imagens $40 \times 120 \times 3$. Os parâmetros treináveis estão destacados (autoria própria).

totalmente conectadas.

2.3.4 USO DE CORES

Em alguns sistemas de visão computacional são usadas imagens em tons de cinza. No caso de redes convolucionais o mais comum é usar imagens coloridas. O primeiro motivo para isso é que as imagens e vídeos são normalmente captadas com cores, então o processo de detecção teria que incluir uma operação de conversão para escala de cinza.

Pode-se demonstrar que todos os métodos de conversão lineares de RGB para escala de cinzas, como:

$$G = 0.299R + 0,587G + 0.114B \tag{11}$$

podem ser representados como uma convolução 1×1. Visto que as convoluções são implementadas de forma bastante eficiente nas bibliotecas de redes neurais convolucionais, algumas vezes até usando aceleração por hardware, não necessariamente haverá economia durante a execução durante a execução da rede neural. No pior caso, quando a cor não é útil, a CNN pode fazer essa conversão de forma eficiente, sendo que os coeficientes associados a cada canal, que no exemplo da equação 11 são fixos, podem escolhidos pelo otimizador, potencialmente obtendo uma conversão mais apropriada para o caso específico onde a rede neural está sendo aplicada.

Outro ponto é que os sistemas que representam o atual estado-da-arte para reconhecimento de imagem, como (SZEGEDY et al., 2015a) (HASANPOUR et al., 2016), usam cores, portanto nos casos onde existe *budget* computacional, e particularmente quando é possível usar vários filtros convolucionais na primeira camada, a informação da cor pode ser vantajosa para minimizar os erros da rede neural.

2.3.5 RECONHECIMENTO DE OBJETOS

Existem várias maneiras de se usar redes neurais para reconhecer objetos. Duas das opções envolvem configurar a rede neural como classificador e como sistema de regressão.

Para usar a rede como classificador pode-se incluir na última camada da rede neural um sistema *softmax*, também conhecido como exponencial normalizada. Isso permite que a rede neural seja treinada e, após o treinamento, estime a probabilidade da sua entrada pertencer a cada uma de N classes. A rede neural é implementada usando N saídas, de forma que cada

saída é treinada para representar a probabilidade da entrada ser classificada em uma das classes. Uma camada *softmax* realiza uma normalização das saídas da rede neural, usando a equação:

$$softmax[i] = \frac{exp\left(\widehat{out}[i]\right)}{\sum_j exp\left(\widehat{out}[j]\right)} \tag{12}$$

Para identificar a presença ou não do objeto que está sendo procurado, duas classes seriam definidas: “presente” e “não-presente”, resultando em uma rede neural como mostrada na figura 11.

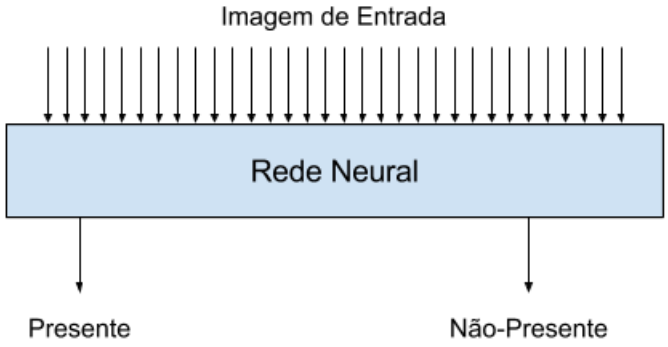

Figura 11: Exemplo de Classificador Logístico. Este exemplo de rede neural possui uma imagem como entrada e possui duas saídas, uma informando a probabilidade da imagem conter o objeto, outra indicando a probabilidade de não conter (autoria própria).

Este modelo pode ser treinado usando modo supervisionado, onde imagens pré-classificadas são fornecidas. A função de perda a ser minimizada pelo otimizador é a função *cross entropy*:

$$\widehat{H} = -\sum_i \widehat{y}_i log(y_i) \tag{13}$$

Onde y_i é o valor da probabilidade manualmente marcada de uma imagem pertencer à classe *i* (no caso, a probabilidade de ser uma imagem que contém o objeto e a probabilidade de não conter) e $\widehat{y_i}$ é o valor estimado pela rede neural. $\widehat{H}$ é o valor estimado de *cross entropy*.

Quando tem-se o objeto desejado centrado na imagem a saída “presente“ deve ser 1 e a outra saída 0. Quando o objeto não é visível em lugar nenhum no campo visual da rede neural os valores devem ser invertidos. Os casos onde o objeto está visível apenas parcialmente devem ser considerados cuidadosamente, produzindo valores intermediários nos rótulos, para evitar que duas imagens parecidas gerem valores muito diferentes na saída da rede neural.

As redes neurais também podem ser usadas para fazer regressão, ou seja, modelar uma função. Uma forma usual de implementação é a regressão L2. Neste tipo de função o otimizador vai minimizar o erro quadrático:

$$E = sum\left((y - \widehat{y})^2\right) \tag{14}$$

Sendo que *sum* é uma função que representa a soma de todos os valores do tensor que recebe como parâmetro. A rede neural pode ser usada para modelar a função P, definida por:

$$y = P(img) = \begin{cases} 1, \text{ se img contém o objeto procurado} \\ t(img), \text{ se objeto está parcialmente visível} \\ 0, \text{ se img não contém} \end{cases} \tag{15}$$

Detectores podem encontrar o objeto que estão procurando parcialmente presentes no seu campo de visão, e esta função precisa tratar desta possibilidade. No caso, a condição de transição, representada por $t(img)$, não está sendo aqui definida, pois cada implementação deve escolher esta função cuidadosamente para integração com o *software* que vem depois da rede neural. Uma coisa que não pode ocorrer é descontinuidade entre duas imagens próximas, especialmente no caso de translação, pois o otimizador teria que encontrar pesos que modelassem a descontinuidade, e mesmo uma pequena imprecisão nestes pesos vai gerar um erro muito grande.

2.3.6 LOCALIZAÇÃO DE OBJETOS

Até agora foi mostrado como usar redes neurais convolucionais para detectar um objeto em uma imagem. Isso envolve apenas determinar se uma imagem possui ou não o objeto de interesse. Localização, ao contrário, requer a determinação das coordenadas do objeto. Em alguns casos pode requerer o cálculo do retângulo envolvente mínimo ou até da curva de perímetro do objeto.

Um sistema de localização pode ser construído a partir de um sistema de detecção. A abordagem mais simples para isso envolve construir um sistema capaz de detectar o objeto de interesse e aplicar este detector à imagem várias vezes a imagem usando “janelas deslizantes”. Se o detector for construído para produzir “1” quando detectar o objeto e “0” caso contrário, então os pontos de máxima são candidatos a centros dos objetos.

Se os objetos puderem variar muito em tamanho pode ser necessário o uso de técnica de aplicação multi escala, onde a janela deslizante também é aplicada a versões reduzidas da imagem. Caso o objeto procurado varie pouco em tamanho isso não é necessário.

Como cada *pixel* da imagem é candidato a centro do objeto que está sendo localizado,

deve-se aplicar o filtro uma vez para cada *pixel*. Se não houver extensão de bordas, o filtro não pode ser aplicado em parte dos *pixels*.

3 TRABALHOS RELACIONADOS

Neste capítulo é feita uma apresentação do atual estado da pesquisa na área deste trabalho. Na seção 3.1 são apresentados trabalhos que aplicam redes neurais convolucionais em imagens, na 3.2 são trabalhos que aplicam *machine learning* a problemas de tráfego, e na 3.3 são trabalhos que usam detecção de placas.

3.1 REDES NEURAIS CONVOLUCIONAIS APLICADAS A IMAGENS

Comparações diretas entre diferentes abordagens para reconhecimento e detecção de imagem podem ser realizadas se um conjunto de dados comum e suficientemente grande for usado. A competição *ILSVRC, ImageNet Large Scale Visual Recognition Challenge* é uma competição que ocorre anualmente desde 2010, e é voltada a este fim. Esta competição permite comparar soluções para diferentes problemas, como classificação de imagens e localização de objetos em um conjunto de dados contendo milhões de imagens manualmente rotuladas entre centenas de categorias. O conjunto de dados está disponível publicamente, e existe um *workshop* anual relativo a competição do ano. As imagens publicadas são separados em um conjunto para treinamento, que está rotulado, e outro conjunto, de teste, cujos rótulos não são publicados, e são usados durante a competição ILSVRC (RUSSAKOVSKY et al., 2015).

O campeão de 2010 foi Lin et al. (2010), usando extrator de *features* HOG e LBP e classificador SVM com 71,8% de taxa de acerto (*top 5*), com taxa de classificação de 52,9%.

O campeão de 2011 na categoria de classificação, o vencedor foi Perronnin et al. (2010), empregando vetores *Fisher* comprimidos e classificador SVM. Na categoria de localização foi Sande et al. (2011), usando busca seletiva baseada em agrupamento hierárquico e detecção usando SIFT, RGB-SIFT e SVM como classificador.

Em 2012 houve a primeira vitória de uma submissão baseada em redes neurais convolucionais. Nas categorias de classificação e localização de imagens o melhor resultado foi de Krizhevsky et al. (2012), usando uma rede neural convolucional contendo 60 milhões de parâ-

metros e 65.000 neurônios, usando cinco camadas convolucionais seguidas por camadas *max-pool* e três camadas totalmente conectadas. Apenas uma terceira categoria, criada naquele ano, denominada *classificação fina* foi vencida por uma implementação que não envolvia CNNs.

Em 2013 o vencedor na categoria de detecção usou uma detector de *features* customizado baseado em SIFT. Os detalhes não foram divulgados. Na categoria de classificação, o mesmo time venceu com uma submissão baseada em CNNs, usando 76 milhões de parâmetros. A terceira categoria do ano foi classificação + localização, vencida por Sermanet et al. (2013) usando CNNs e janelas deslizantes de escala múltipla.

Em 2014 os organizadores decidiram separar os resultados de acordo com múltiplos critérios. Um dos resultados vencedores foi Szegedy et al. (2015a), que é baseada em CNNs, e define uma arquitetura denominada *inception*, na qual várias técnicas são usadas para reduzir o número de parâmetros e aumentar o desempenho de classificação, como usar duas camadas convolucionais 3×3 ao invés de uma camada 1×1. A rede neural final possui 22 camadas com parâmetros, ou cerca de 100 camadas no total, usando 1/12 do número de parâmetros usados por Krizhevsky et al. (2012).

Em 2015 um dos vencedores foi He et al. (2015), que empregou uma rede neural com profundidade 152. Segundo o *paper*, redes neurais com essa profundidade, quando treinadas da maneira convencional, produzem resultados piores que redes neurais muito mais rasas. Um método foi proposto para fazer o treinamento, que envolve treinar uma secção da rede neural e progressivamente aumentar o tamanho da rede neural adicionando camadas antes e depois do trecho treinado, sendo que essas camadas adicionadas estão configuradas para realizarem o equivalente a uma função identidade, não afetando o resultado do treinamento que já foi realizado. O método de treinamento foi demonstrado no conjunto de dados CIFAR em um modelo convolucional com 1000 camadas.

3.2 APRENDIZADO DE MÁQUINAS PARA MONITORAMENTO DE TRÁFEGO

Existem sistemas com variadas capacidades aplicadas a monitoramento e fiscalização de tráfego. Os sistemas que usam visão computacional frequentemente aplicam técnicas de aprendizado de máquinas.

Alguns sistemas precisam identificar a placa veicular. Para tal, técnicas de OCR precisam ser empregadas. Muitos sistemas, como Qadri e Asif (2009) e Kranthi et al. (2011) usam uma sequência que envolve pré-processamento, localização da placa, segmentação das letras (ou números) e reconhecimento dos caracteres. Por mais que os primeiros passos não usem

necessariamente aprendizado de máquinas diretamente, elas incluem OCR, que normalmente é implementado usando alguma técnica que envolve treinamento. No primeiro caso citado o uso é indireto, através de um módulo de *software* pré treinado que o autor usou. No segundo caso o treinamento do sistema de *machine learning* foi feito pelo próprio autor.

A solução apresentada em Kim et al. (2000) também usa a mesma sequência de passos para fazer a leitura da placa, porém usa redes neurais para fazer a segmentação e SVM para fazer reconhecimento de caracteres. Para a segmentação o autor usou duas TDNNs, que é uma rede neural não convolucional que fornece para a primeira camada oculta não apenas o valor atual da entrada, mas também o valor da entrada no processamento em $t-1$, $t-2$, ..., $t-n$. O autor aplica a rede neural linha por linha nos *pixels* da rede neural, e outra rede neural coluna por coluna, processando com um algoritmo escrito manualmente o resultado das duas redes neurais para obter as regiões envolventes dos segmentos que contém os dígitos da placa.

Outra área onde estratégias de aprendizado de máquinas pode ser usado é predição de tráfego. Em Guo et al. (2012) foi proposto um método para fazer predição em curto prazo para condições normais e anormais de tráfego. A técnica proposta envolve o uso de kNN (*k-Nearest Neighbours*) aplicado aos dados de um SSA (*Singular Spectrum Analysis*). O SSA é usado para suavizar os dados antes deles serem fornecidos para o kNN, que faz a predição do tráfego.

3.3 DETECÇÃO DE PLACAS DE VEÍCULOS

Em Luvizon et al. (2016) foi proposto um método para estimar velocidade de veículos pela detecção e rastreamento de placas veiculares. A etapa de detecção de placas usa *SnooperText* (MINETTO et al., 2010) configurado especificamente para esta tarefa. *SnooperText* é um detector de texto que opera segmentando o texto usando *toggle mapping*, depois filtrando caracteres usando regras de forma, três descritores e quatro classificadores SVM, depois agrupados com regras de forma e distância e finalmente filtrados usando T-HOG e mais um classificador SVM. A detecção de placas ocorre usando detector de movimentos

Anagnostopoulos et al. (2008) faz um *survey* de várias técnicas de reconhecimento de placas de trânsito. Neste artigo captura em "tempo real" é considerado um sistema que processa cada *frame* em 50ms. Este *survey* encontrou sistemas que tipicamente fazem pré processamento, alguns usando imagens binárias, alguns usando tons de cinza e outros usando cores.

Entre as abordagens que usam imagens binárias encontram-se sistemas baseados em detecção de bordas que, apesar de muito rápidos, geram muitos falsos positivos. Um sistema usa *Connected Component Analysis*, que analisa componentes conectados quanto a sua geometria

(dimensões e área) para determinar se cada componente é uma placa de trânsito. O uso do operador *Snobel* é bastante comum entre os artigos pesquisados para produzir resultado final dentro do limite de 50ms.

Ainda de acordo com Anagnostopoulos et al. (2008), a categoria de abordagens que encontram mais soluções é a que aplica tons de cinza. Várias abordagens usam contagem do número de variações abruptas de contraste em um eixo, tipicamente o horizontal, para localizar placas. Este tipo de algoritmo pode operar em uma a cada *N* linhas da imagens, sendo muito econômico em tempo de CPU, porém é simplista demais para operar em vários cenários. Processamento estatístico de bordas pode ser usada focando nas letras para operar bem quando o contorna da placa não é clara. O uso de *quadtrees* hierárquicos foi proposto, no qual cada quadrante é novamente dividido se tiver bastante variação de contraste. Segmentos contíguos são agrupados se o brilho deles for muito diferente ou muito próximo. Cada segmento recebe um escore de acordo com seu tamanho e o escore dos blocos que o compõe, que também é baseado em tamanhos. Os melhores *strips* (*quadtrees* sucessivos) são selecionados. Este algoritmo é bastante robusto a condições de iluminação e tem boa taxa de acertos. Uso de janela deslisante, nas quais a média e o desvio padrão são calculados e usados diretamente contra um limiar também foram usados com sucesso. *Wavelet transform* foi aplicado para localizar placas, mas o método se mostrou muito sensível a variações de distâncias e das características das lentes.

Tentativas de uso das informações de cor utilizadas nos diversos países foram realizadas, porém não se mostravam estáveis em condições naturais de iluminação, pelo fato de que a impressão das cores varia de acordo com a luminosidade. As tentativas envolvem desde classificação *pixel* a *pixel* das cores até uso de lógica *fuzzy*, redes neurais com taxas variadas de acerto, de 75% a 98%.

O uso de câmeras e iluminação infravermelha foram demonstradas como sendo capazes de produzir sistemas com taxa de acerto de 99.3%. Baixo número de *pixels* (baixa resolução) foi demonstrado como tendo efeito negativo em todos os sistemas testados.

Em Jain e Kundargi (2015) um método de reconhecimento de placas usando redes neurais é proposto, porém a etapa de localização usa filtro *Sobel* para detectar bordas e *connected component analysis*.

4 LOCALIZAÇÃO DE PLACAS VEICULARES USANDO REDES NEURAIS CONVOLUCIONAIS

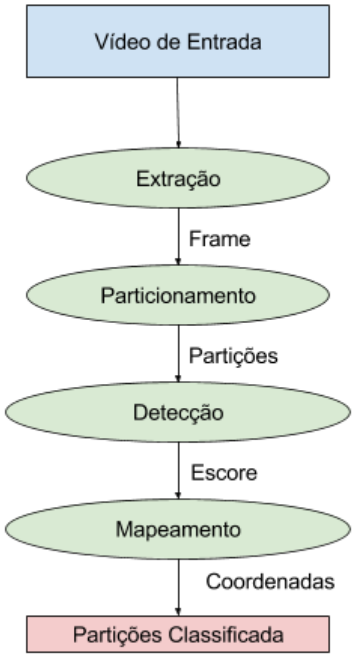

Figura 12: Etapas do método proposto para localização de placas veiculares (autoria própria).

O método proposto para a localização de placas envolve uma série de etapas, conforme ilustrado na figura 12. O processo "extração" é responsável por obter os *frames* do vídeo. Um *frame* é equivalente a uma imagem 2D, e pode ser representado por tensor de dimensões HWC (altura, largura e canais de cor). Todos os *frames* de um mesmo vídeo geram tensores do mesmo tamanho. O processo "particionamento" toma como entrada um *frame* do vídeo e gera múltiplas sub imagens aqui denominadas "partições", geradas através de um recorte retangular de *pixels* contíguos. As partições têm dimensões compatíveis com a rede neural que será usada no passo seguinte. A etapa denominada "detecção" é, mais especificamente, uma rede neural artificial convolucional profunda treinada para modelar uma função escalar cujo valor, denominado "escore", é determinado pela presença e posição, ou não presença, de uma placa veicular.

A etapa "mapeamento" recebe todos os escores calculados para todas as partições da

imagem e calcula a partir delas as coordenadas dos centros das placas. Quando uma placa está perfeitamente centrada em uma partição esta terá valor 1, enquanto todas as vizinhas terão o valor 0. Quando o centro da placa está entre duas partições então ambas terão valor superior a zero. A implementação usada aqui considera apenas o maior valor. A posição do centro da placa é estimado como sendo a posição do centro da partição onde ela foi detectada.

O método aqui proposto inclui algumas opções na implementação do processo supra-descrito. Elas dizem respeito:

- ao tamanho das partições de imagem a serem recortadas e, por consequência, ao tamanho da entrada da rede neural;
- à escolha do *stride* das partições da rede neural;
- à escolha da função que a rede neural está modelando;
- à escolha da arquitetura e dos hiperparâmetros da rede neural.

4.1 TAMANHO DAS PARTIÇÕES

Figura 13: Exemplo de três tamanhos de partições. Em (a) a partição possui um grande campo visual, porém tem baixa precisão. Em (b) há um campo visual muito pequeno para permitir que a rede neural opere corretamente. A terceira é apropriada, pois é relativamente pequena e inclui a placa inteira e vários *pixels* adicionais, permitindo que a rede neural aprenda o contraste entre placa e o fundo (autoria própria).

O tamanho das partições define as dimensões do tensor da rede neural, portanto o tamanho do seu "campo visual". Quanto maior o campo visual da rede neural, mais dados ela vai ter para processar, porém menor será a precisão. Se o campo visual for muito pequeno, como na figura 13b a precisão seria maximizada, mas a área não é suficiente para a rede neural operar corretamente. Se o campo visual for muito grande, como na figura 13a, a rede neural terá mais características para usar, mas a precisão da localização será muito prejudicada.

De forma geral, as dimensões das partições devem ser minimizadas, mas deve-se garantir que a placa caiba no seu interior, com um pouco de sobra. As figuras 13c e 14 mostram um bom tamanho. A região adicional fora da placa vai servir para a rede neural aprender que existe um contraste de *features* entre o interior e o exterior da placa. Este contraste vai ser usado não só para identificar quando existe uma placa, mas também para excluir muitos casos de falsos positivos, como *banners* e outros textos que podem ser encontrados em veículos.

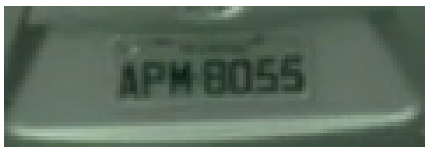

Figura 14: Exemplo de partição com bom tamanho. (autoria própria).

4.2 *STRIDES* DAS PARTIÇÕES

A abordagem que está sendo proposta envolve usar um detector para construir um localizador. Uma implementação ingênua desta abordagem seria treinar o detector para detectar placas de trânsito que estejam centradas na sua entrada e aplicar este detector como descrito na seção 2.3.6, usando *stride* 1×1. Isso iria requerer aplicar a rede neural centrada em cada *pixel* da imagem. Se for feita a aplicação de um detector com campo visual $D_1 \times D_2$ em um *frame* com dimensões $F_1 \times F_2$ com *stride* 1×1 sem estender as bordas pode-se demonstrar que o detector terá que ser aplicado N vezes, onde:

$$N = (F_1 - D_1 + 1) \cdot (F_2 - D_2 + 1) \tag{16}$$

Para um detector com dimensões HW de 40×120 em uma imagem 480×768 seriam geradas 286.209 partições, e cada uma delas precisaria ser aplicada ao detector, o que é proibitivo.

Para resolver este problema propõe-se o uso de um *stride* de $50\% \times 50\%$ do tamanho da partição. Isso significa que duas amostras consecutivas (lado-a-lado) tem os seus centros a uma distância igual a sua largura sobre dois, gerando partições como as ilustradas na figura 15.

Uma propriedade dessa escolha é que o centro de uma partição de imagem está contido no perímetro de todas as partições vizinhas. Esta propriedade é justamente o objetivo do *stride* de 50%, conforme será descrito na seção 4.3.

O número de partições na direção da altura será T_H, e na largura será T_W, sendo:

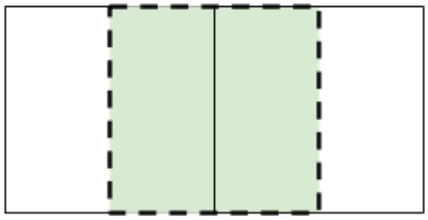

Figura 15: Ilustração de 3 partições sucessivas. Observar que o centro de uma partição está contido no perímetro das partições vizinhas (autoria própria).

$$T_H = \left\lfloor \frac{2F_1}{D_1} - 1 \right\rfloor \tag{17}$$

$$T_W = \left\lfloor \frac{2F_2}{D_2} - 1 \right\rfloor \tag{18}$$

E o número de partições a serem classificadas será:

$$N = \left\lfloor \frac{2F_1}{D_1} - 1 \right\rfloor \cdot \left\lfloor \frac{2F_2}{D_2} - 1 \right\rfloor \tag{19}$$

Tomando o mesmo caso calculado anteriormente (detector 40×120 em *frames* 480×768 sem extensão de borda) com a estratégia proposta de particionamento o número de partições a serem classificadas cai de 286.209 para $23 \cdot 11 = 253$.

4.3 FUNÇÃO A SER MODELADA

A rede neural vai ser treinada para modelar o valor de uma função P que leva um tensor que representa uma partição do *frame* a um real no intervalo $[0;1]$:

$$P : S \to [0;1] \in \mathbb{R} \tag{20}$$

A função deve produzir o valor 1 quando existe uma placa centralizada no seu campo visual, como ilustrado na figura 16a. A função mantém o valor 1 até que a placa esteja como em 16b, precisamente na metade da distância para a posição 16c, quando a placa está com o seu centro, na borda da imagem de entrada. Esta imagem ilustra apenas o efeito da placa estar fora de centro em uma dimensão.

A função proposta foi construída baseada na forma com que a placa veicular passa de uma partição para a partição vizinha. A função deve possuir as seguintes características:

Figura 16: Imagem ilustrando como a posição da placa afeta a função. O valor de saída da função depende da posição do centro da placa que está sendo vista. Se estiver como em (a), centralizada, a função produzirá o valor 1. Se estiver como em (c), onde o centro da placa está no perímetro da partição, a função deve produzir o valor 0. No ponto (b), onde está precisamente entre os dois pontos anteriores, também deve produzir o valor 1. Nesta ilustração a placa está sempre centralizada horizontalmente (autoria própria).

- a função é contínua quando uma placa é deslocada de forma contínua na entrada por qualquer caminho;
- quando a placa veicular está no centro de uma partição a função de saída deve ser 1, e quando está no centro da partição vizinha deve ser 0;
- uma, e apenas uma partição deve possuir valor 1 como consequência da presença da placa, exceto nos pontos críticos, na vizinhança dos quais a função é menor que 1.

A figura 17 ilustra os pontos principais quando uma placa é movida horizontalmente entre duas partições.

A figura 18 mostra o valor da função para duas partições vizinhas à medida que a placa é movida. Pode-se observar que apenas uma das partições possui valor 1 por vez, exceto no ponto cuja vizinhança é menor que 1. Também observa-se que quando a placa está centrada em uma partição, a função possui valor zero na outra partição.

Na figura 19 observa-se o valor de uma função quando a placa está alinhada na altura e é deslocada horizontalmente. Como a abscissa é a distância normalizada entre os centros obtém-se o valor 0,5 quando o centro da placa está a uma distância igual a metade do tamanho da partição para a direita, como ilustrado na figura 17c.

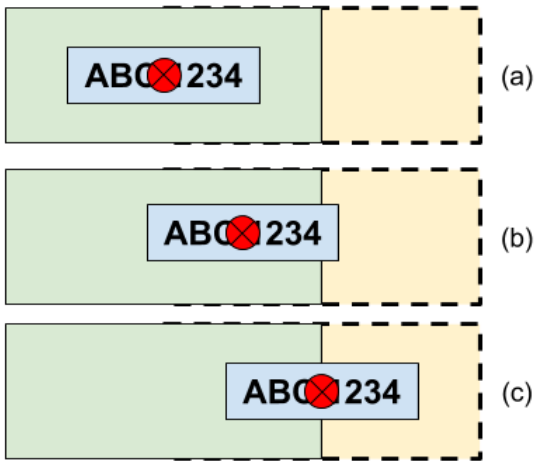

Figura 17: Placa veicular sendo movida entre duas partições. Ilustração dos pontos críticos para a definição da função a ser modelada pela rede neural, mostrando duas partições vizinhas, uma pintada de verde e o outro amarelo com borda pontilhada. Em (a) a placa está centrada na partição da esquerda e no perímetro da partição da direita. Em (b) o centro da placa está no meio caminho entre os centros dos duas partições. Em (c) o centro da placa está no centro da partição da direita e perímetro da partição da esquerda (autoria própria).

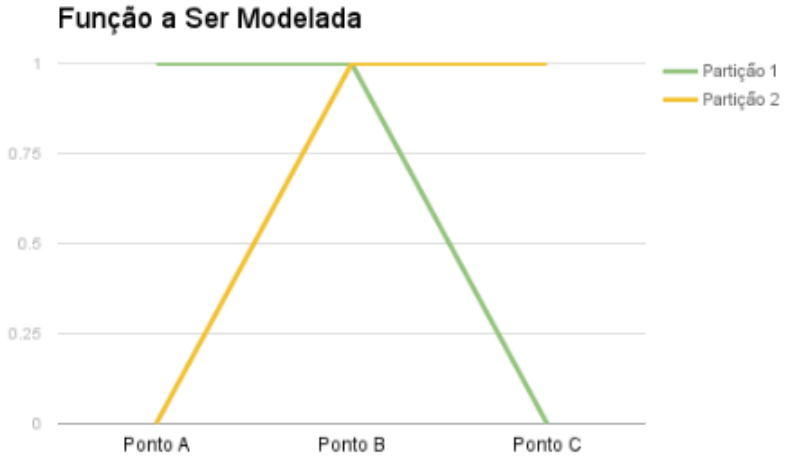

Figura 18: Valor da função em duas partições a medida que a placa se move entre elas. Observa-se que valor da saída da função para duas partições a medida que a placa é movida do centro da partição 1 (a da esquerda) para a partição 2 (a da direita) (autoria própria).

No caso de uma placa alinhada na direção y (distância em y é 0), e sendo deslocada apenas em x, variando a distância normalizada Δx, a função proposta é:

$$f_x(\Delta x) = \begin{cases} 1, \text{ se } |\Delta x| \leq 1/4 \\ \frac{1/2-|\Delta x|}{1/2-1/4}, \text{ se } 1/4 < |\Delta x| < 1/2 \\ 0, \text{ se } |\Delta x| \geq 1/2 \end{cases} \tag{21}$$

Da maneira semelhante, quando a distância em x é zero e a distância normalizada Δy é

Figura 19: Valor da função a modelar, de ∞ a $+\infty$. Observa-se que o valor da variável dependente a medida que a placa é deslocada desde $-\infty$ até $+\infty$ da esquerda para a direita, enquanto a mesma está centrada na altura. A abscissa é a distância normalizada entre os centros da partição e da placa (autoria própria).

livre:

$$f_y(\Delta x) = \begin{cases} 1, \text{ se } |\Delta y| \leq 1/4 \\ \frac{1/2-|\Delta y|}{1/2-1/4}, \text{ se } 1/4 < |\Delta y| < 1/2 \\ 0, \text{ se } |\Delta y| \geq 1/2 \end{cases} \tag{22}$$

Quando usado com as direções x e y livres a função a ser estimada pela rede neural é:

$$f = f_x \cdot f_y \tag{23}$$

O gráfico *3D* desta função está representado na figura 20, mostrando o efeito simultâneo dos eixos x e y. Observa-se que quando a placa está com o seu centro a uma distância normalizada inferior 1/4 em x e em y simultaneamente, a função vai produzir o valor 1. O gráfico não possui descontinuidades, portanto pertence a classe C_0, mas não pertence à classe C_1, por que as derivadas não são contínuas. Não seria apropriado usar uma função descontínua por que isso geraria problemas durante o treinamento.

Quando a distância normalizada supera $^1/_2$ em qualquer direção o valor da função é 0.

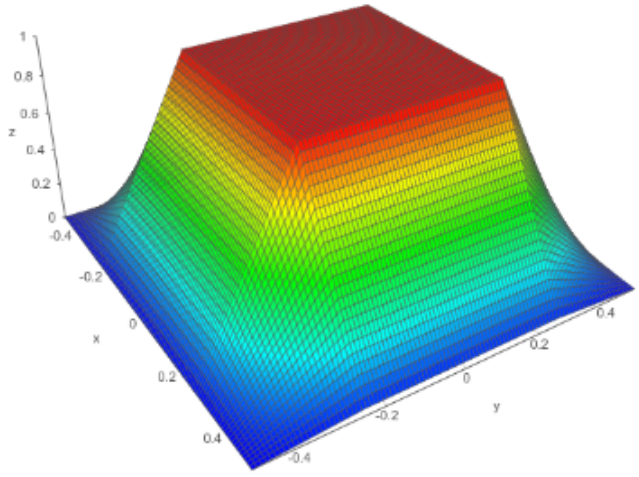

Figura 20: Gráfico 3D da função que a rede neural deve modelar. As abscissas representam a distância normalizada entre o centro da partição e da placa em cada direção (autoria própria).

4.4 ARQUITETURA DA REDE NEURAL

A arquitetura da rede neural, incluindo seus hiperparâmetros, deve ser escolhida de forma a balancear desempenho de classificação com tempo de propagação. Se a rede neural requerer muitas operações, pode aumentar o desempenho de classificação, mas o tempo de processamento vai aumentar. Idealmente a rede neural deve ser restrita a um tamanho que permita que todos as partições da imagem sejam classificadas de forma que o sistema como um todo entregue a taxa de *frames* desejada para a resolução do vídeo necessária e hardware disponíveis onde a solução vai ser implantada.

Na implementação da rede neural convolucional existem otimizações que podem ser feitas para reduzir o número de operações enquanto mantendo ou reduzindo pouco a capacidade da rede neural de executar a operação para a qual vai ser treinada. Estas otimizações são um campo ativo de pesquisa, sendo relevantes os *papers* produzidos como resultado da competição *ImageNet Large Scale Visual Recognition Competition (ILSVRC)*. Os *papers* (SZEGEDY et al., 2015a) e (SZEGEDY et al., 2015b), por exemplo, detalham várias substituições que podem ser feitas para este fim.

5 IMPLEMENTAÇÃO E EXPERIMENTOS

Neste capítulo será apresentada uma implementação do método descrito no capítulo 4. Esta implementação será usada para medir o desempenho do método proposto segundo métricas padronizadas, além de métricas adicionais.

Na seção 5.1 será apresentada a arquitetura da implementação. Na seção 5.2 serão apresentadas tentativas mal sucedidas que foram realizadas antes da versão final, mostrando por quê elas falharam. A seção 5.3 são apresentadas as tecnologias usadas na implementação, e na 5.4 os recursos de hardware. O capítulo 5.5 apresenta o software como foi implementado.

5.1 ARQUITETURA GLOBAL

Redes neurais em geral requerem grande quantidade de exemplos de treinamento para evitar *overfitting* (HAWKINS, 2004). É crucial para o sucesso de uma implementação acesso a dados ou a capacidade de produzi-los.

Durante a faze conceitual optou-se por dar um grande enfoque no treinamento da rede neural. Como não existe nenhuma fonte pública de imagens etiquetadas de placas de carro com as características necessárias optou-se por gerar estes dados a partir de imagens de vídeo. Para tal, dois módulos foram desenvolvidos: um de marcação manual de placas em vídeo e um para geração desses dados. Usando os dois é possível gerar exemplos rotulados em quantidade suficiente no formato requerido pela rede neural. O treinamento e os testes requereram o desenvolvimento de outros dois outros módulos: o primeiro treina a rede neural, o segundo reproduz o vídeo enquanto aplica a rede neural treinada, mostrando as placas identificadas. Isso resulta em quatro módulos de *software* principais. O relacionamento entre eles é mostrado na figura 21.

A implementação de cada um destes módulos mudou durante a história do projeto, porém as suas funções básicas permaneceram as mesmas.

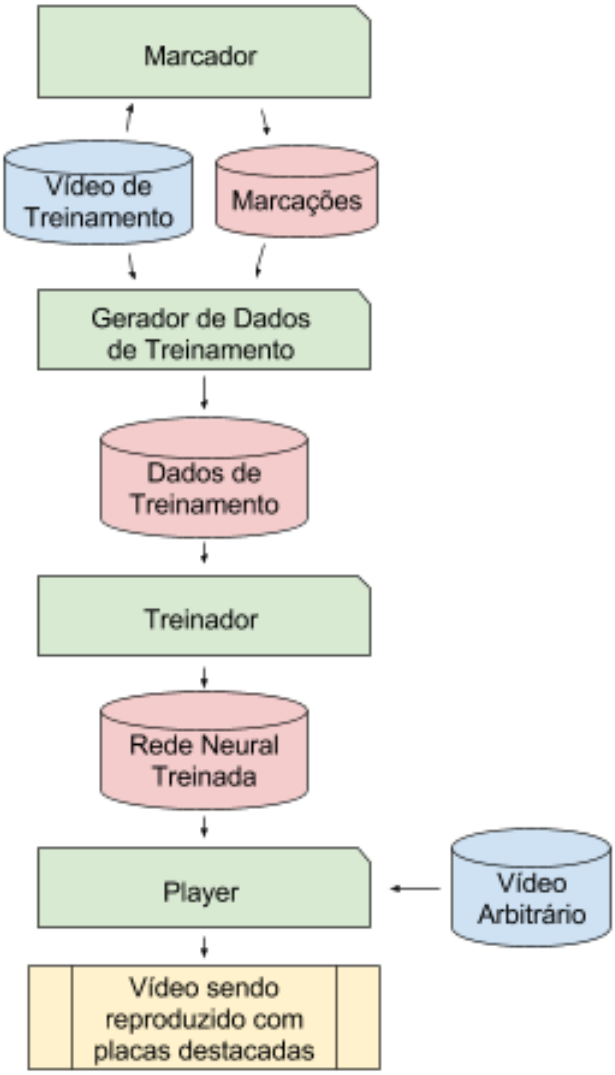

Figura 21: Arquitetura Global da Implementação. Diagrama ilustrando os quatro componentes de *software* em verde, os vídeos de entrada em azul, as informações intermediárias em vermelho e a saída do sistema em amarelo (autoria própria).

5.2 ABORDAGENS ANTERIORES

Durante o desenvolvimento do projeto algumas tentativas mal sucedidas foram feitas antes da versão final ter sido completada. O *software* chegou a ser implementado por completo três vezes, sendo que só produziu resultados satisfatórios na última.

A ideia inicial para este projeto era o uso de redes neurais não-convolucionais aplicadas diretamente aos *pixels* das imagens. Para tal foi escolhida a biblioteca *Neuroph*, por ter sido implementado na linguagem Java e pela sua grande flexibilidade. A solução foi totalmente implementada conforme arquitetura ilustrada na figura 21.

Essa implementação usava imagens em tons de cinza e partições com dimensões HW

32×100 aplicadas em uma rede neural totalmente conectada com 3200 entradas e uma saída. O particionamento era feito usando *stride* de 100%, ou seja, duas partições vizinhas tinham o perímetro de um dos seus lados em comum.

Esta biblioteca foi logo descartada porque o tempo de treinamento era muito longo, impedindo a busca eficiente de configuração das camadas ocultas que gerasse o resultado desejado. Testes com topologias mais complexas, com maior largura e profundidade nas camadas intermediárias, chegavam a passar de uma semana de execução quando rodadas em um notebook *Core i7*. Não houve nenhuma configuração encontrada com desempenho de classificação aceitável.

Acreditando que seria possível resolver o problema encontrando os hiperparâmetros corretos da rede neural, e sabendo que isso requeriria múltiplos experimentos com topologias diferentes, adotou-se a biblioteca *Encog*, devido ao melhor desempenho de treinamento. Essa adaptação requereu reescrever boa parte código. Apesar do treinamento rodar a uma taxa de imagens 10 vezes maior que a biblioteca anterior, também não foi encontrada topologia com desempenho aceitável. A figura 22 mostra a evolução do treinamento para algumas sessões. Entre as listadas a que teve melhor desempenho usava 3200 neurônios na camada de entrada, então 172, 137, 109, 87, 69, 55, 44, 35 e 28 neurônios nas camadas ocultas, terminando com uma saída.

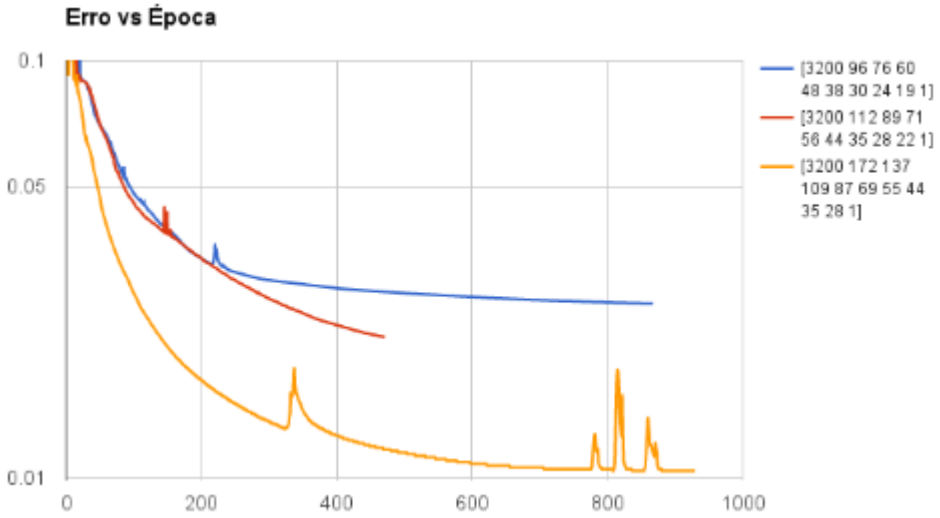

Figura 22: Erro de treinamento de redes neurais não-convolucionais. A ordenada refere-se ao erro L2 calculado sem a raiz quadrada e suavizado usando *exponential smoothing*. A legenda refere-se ao número de neurônios em cada camada totalmente conectada (autoria própria).

Quando o *player* era usado com as redes neurais resultantes desse treinamento o resultado era muito ruim, com quantidade muito grande de falsos negativos. Eventualmente descobriu-se que a forma com que os exemplos de imagens que não continham placa estava

sendo coletada possuía uma grande quantidade de imagens muito parecidas. Aparentemente a rede neural não estava conseguindo generalizar as características de uma placa veicular e, ao invés disso, encontrou parâmetros para as camadas totalmente conectadas que identificavam corretamente alguns destes exemplos, o que acabava reduzindo o erro de treinamento até certo ponto. A redução das imagens parecidas no conjunto de treinamento de entrada só foi feito depois das redes neurais totalmente conectadas terem sido abandonadas.

O esquema de particionamento das imagens foi mudado de 100% para 50%, que é o valor usado agora. A função que a rede neural modela foi modificada várias vezes, mas o resultado final, conforme visto no *player*, continuava ruim.

Após alguns meses a abordagem foi abandonada. Foi feita uma pesquisa sobre o assunto, e descobriu-se a existência de redes neurais convolucionais. Infelizmente a biblioteca *Encog* e a *Neuroph* não suportam este tipo de topologia. Também não foram encontradas bibliotecas com grande base de usuários em Java.

Neste ponto foi escolhida a biblioteca recém-lançada *TensorFlow*. Inicialmente a preparação dos dados de treinamento continuou sendo feita pelo código em Java que já havia sido escrito, enquanto o código de treinamento e execução foram reescritos em Python, que era a única linguagem suportada quando essa migração foi realizada.

O uso de duas linguagens de programação estava eliminando várias oportunidades de reuso de código. Eventualmente os *softwares* de marcação e preparação de dados de treinamento foi reescrito em Python.

Desde os primeiros experimentos a abordagem usando redes neurais foi muito bem sucedida. A primeira abordagem foi usando regressão logística. No entanto os exemplos de treinamento foram etiquetados usando apenas valores 0 e 1. Por não saber exatamente como mapear uma placa parcialmente visível para probabilidades, foi feita a migração para um modelo de regressão, onde uma função suave pode ser definida para ser imitada pela rede neural.

5.3 ESCOLHA DE TECNOLOGIAS

A versão final da *deep convolutional neural network* usa o *framework TensorFlow* do Google. No *TensorFlow* usa-se a linguagem Python para descrever um ou mais grafos onde os nós são operações e as arestas são tensores. Uma vez que o grafo esteja totalmente definido podem-se fornecer dados e solicitar o cálculo de qualquer quantidade de nós. A execução em si ocorre em um *runtime* de alto desempenho escrito em C++ e CUDA, que pode ser distribuído entre múltiplas máquinas e pode rodar em CPUs e GPUs.

A tecnologia foi escolhida por ser *open source*, e por ser a ferramenta utilizada pelo Google. Esta biblioteca é fácil de usar para pequenos projetos, como este, e poderosa para poder escalar para sistemas com múltiplos computadores usando GPUs de alto desempenho. O *TensorFlow* permite usar Python 2 ou Python 3, sendo que a versão 3 foi escolhida para o projeto.

A versão 3 do OpenCV, bem como seu *wrapper* Python foram adotados para leitura e exibição do vídeo, e para algumas operações de manipulação de imagens e para a construção da interface com usuário. O OpenCV tem funcionalidades suficientes para exibir uma janela contendo uma imagem e capturar eventos de teclado e mouse. Isso é suficiente para toda a interface gráfica.

Tanto o *wrapper* Python do OpenCV quanto o *TensorFlow* representam dados numéricos, como tensores, usando uma biblioteca numérica para Python chamada *NumPy*. Ela foi usada para várias operações, como extrair sub imagens dos *frames* de vídeo, após terem sido lidos pelo OpenCV, e fornecer estes dados para o *TensorFlow*.

Para representar as marcações que o usuário faz nos vídeos, para identificar onde estão as placas, foi usado o formato JSON. Existem boas bibliotecas em várias linguagens, inclusive Python e Java, o que garante que os dados poderão ser lidos facilmente.

O Linux foi o único sistema operacional usado, por ser *open source*, gratuitamente acessível e poder ser usado em produtos comerciais, caso surja a demanda. A distribuição escolhida foi Arch Linux, apesar de não ser oficialmente suportado pelo projeto *TensorFlow*, por possuir a filosofia de empregar sempre a versão mais atual de todos os componentes, como kernel. O suporte ao *TensorFlow* é oferecido pela comunidade AUR (Arch User Repository), que permite a instalação a partir do código-fonte do *TensorFlow* mesmo antes de uma versão oficial ser liberada.

O sistema de controle de versões usado foi o GIT.

Para edição de código foi usado exclusivamente *Vim*, de forma que não houve necessidade de nenhuma licença para o desenvolvimento.

5.4 RECURSOS DE HARDWARE

O único recurso de hardware que foi necessário para este projeto foi um computador, sendo que dois foram usados:

- Um notebook Core i7-3632QM com 4 cores (2 threads por core), 16 GiB de RAM;

- Um notebook Core i5-4210U com 2 cores (2 threads por core), 8 GiB de RAM, placa de video NVidia GT-750M com 2 GiB de RAM compatível com *TensorFlow*.

O desenvolvimento do projeto foi quase todo feito usando o Core i7. Assim que o suporte a GPU foi incluída no código o notebook Core i5 começou a ser usado concomitantemente.

A CPU do computador Core i7, quando usado para treinamento, rapidamente atinge 85° e o desempenho do treinamento reduz consideravelmente, como consequência da CPU estar atuando para limitar a potência dissipada. Após o computador rodar por alguns meses desta maneira o computador deixou de ser confiável, travando com frequência e muitas vezes se recusando a ligar com sintomas aleatórios. No caso de treinamento prolongado, como foi feito neste projeto, pode ser necessário conferir a temperatura na qual os componentes estão rodando, para evitar danos.

5.5 IMPLEMENTAÇÃO DOS MÓDULOS DE *SOFTWARE*

Nesta seção está detalhada a implementação de cada módulo de *software*.

5.5.1 MARCADOR

Figura 23: Interface com usuário do *Marcador*. A tela está mostrando 3 veículos marcados (autoria própria).

Tabela 1: Arquivos de vídeo usados durante o desenvolvimento

Vídeo	Resolução	FPS	Tamanho	Duração
video1.avi	480×768	25,0	141 MiB	8:27
video2.avi	1080×1920	25,0	1,0 GiB	36:16
video3.avi	1080×1920	25,0	465 MiB	8:02

O *software* marcador foi construído como um script Python que recebe o nome de um arquivo de vídeo como parâmetro. Ele permite:

1. Reproduzir o vídeo;
2. Pausar a reprodução do vídeo
3. Avançar / voltar *frame-a-frame*;
4. Saltar para um *frame* digitando-se o número dele;
5. Quando o vídeo está em pausa, permite clicar no vídeo para adicionar um marcador de placa;
6. Usar o teclado para mover, rotacionar e alterar o marcador, de forma que ele circule corretamente a placa sendo marcada;
7. Digitar o número da placa veicular

Como pode ser visto na figura 23, o *software* marcador permite marcar placas de carros e motos, e é possível identificar o número da placa. Isto tem como objetivo permitir que as mesmas marcações possam ser usadas futuramente para fazer OCR das placas. As marcações são salvas em um arquivo JSON.

Quando um *frame* possui pelo menos uma placa veicular marcada, todo o resto da imagem vai ser considerado pelo *software* de treinamento como região sem placa. Por isso, se uma placa for marcada em um *frame*, devem-se marcar todas.

Todo o desenvolvimento foi feito usando três vídeos. As características dos vídeos estão listadas na tabela 1, e um *frame* de cada vídeo está mostrado na figura 24.

A tabela 2 mostra a quantidade de marcações feitas em cada vídeo. A maior parte das marcações foram feitas no *video1*. Observar que existem dois conjuntos de marcações neste vídeo. Uma delas, denominada *testes*, não foi usada para treinamento, pois está reservada para

Figura 24: Um *frame* de cada vídeo usado durante o desenvolvimento. (autoria própria).

Tabela 2: Marcações feitas em cada vídeo

Vídeo	Quadros Marcados	Placas Marcadas
video1.avi	191	233
video1.avi (testes)	90	92
video2.avi	71	96
video3.avi	0	0
TOTAL	352	421

testar o desempenho do modelo. Os dados de treinamento estão todos antes do *frame* número 8000, e todos os dados de teste vêm depois.

5.5.2 GERADOR DE DADOS DE TREINAMENTO

O *software* gerador de dados de treinamento também recebe como parâmetro o nome do arquivo de vídeo onde vai operar, e vai abrir este arquivo de vídeo e as marcações feitas nele. Este *software* roda de modo não-interativo, terminando sem intervenção do usuário quando a geração do conjunto de treinamento está concluída. A interface gráfica está ilustrada na figura 25

O conjunto de treinamento é armazenado em formato binário com registros de tamanho fixo. Cada registro contém a imagem seguida de um rótulo. A imagem é codificada no formato HWC usando 1 byte por canal de cor, sendo a cor na ordem RGB. O rótulo representa o valor que a rede neural deve aprender (número ponto-flutuante de 0 a 1), mas é codificado como um inteiro de 8 bits sem sinal. Como a imagem é $40 \times 120 \times 3$ e o rótulo tem 1 byte, então cada registro possui 14401 bytes.

Este *software* lê a lista de *frames* marcados em uma ordem aleatória. Então vai coletar exemplos de placas veiculares centradas e fora de centro, e exemplos de imagens que não contêm placas, e vai salvá-las no arquivo de saída. O rótulo que vai ser salvo com a imagem é o

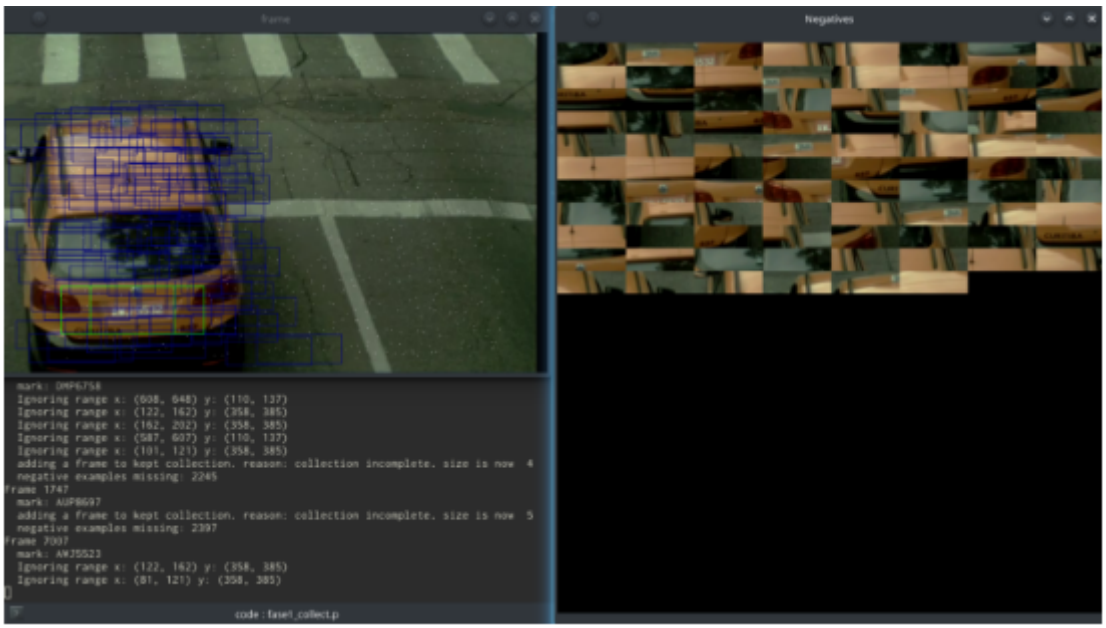

Figura 25: Interface gráfica do gerador de dados. A imagem da esquerda mostra o quadro que está sendo processado e mostra as regiões onde os exemplos negativos estão sendo coletados. A imagem da direita mostra os próprios exemplos negativos que estão sendo enviados para o arquivo (autoria própria).

resultado da equação 23 da seção 4.3, que a rede neural precisa aprender a reproduzir a partir dos dados dos pixels.

Tomando uma placa nas coordenadas $y \times x$, o *software* primeiro gera um recorte centrado nessas coordenadas e com o tamanho 40×120. Este tamanho é definido em um arquivo de configuração. Todos os exemplos coletados desta forma recebem o rótulo 1 (codificado como 255).

Para obter os exemplos de placas não centralizadas são coletados recortes com dimensões 40×120 de aproximadamente 4 em 4 *pixels* dentro de uma região próxima da placa. O algoritmo avança precisamente na taxa de 4 *pixels*, porém adiciona um número aleatório de -2 à 2 em cada um deles. Isso tem como objetivo evitar fornecer dados muito regulares para a rede neural para impedir que ela infira alguma regra relacionada a essa regularidade. A região onde se coletam estes exemplos é tal que o centro fica dentro de uma região 40×120 centrada na placa do veículo, como ilustrado na figura 26.

Quando o centro de uma placa veicular está localizada dentro da região delimitada a função vai produzir valores maiores que zero. Quando o centro da placa está fora dessa região a função vai vai produzir valor zero. Por mais que alguns pixels da placa estejam dentro desta partição, a partição vizinha possui mais pixels da mesma placa. O ponto (c) na figura 26 mostra o limiar que está a $^1/_4$ de distância do centro na direção da largura. Neste ponto a função

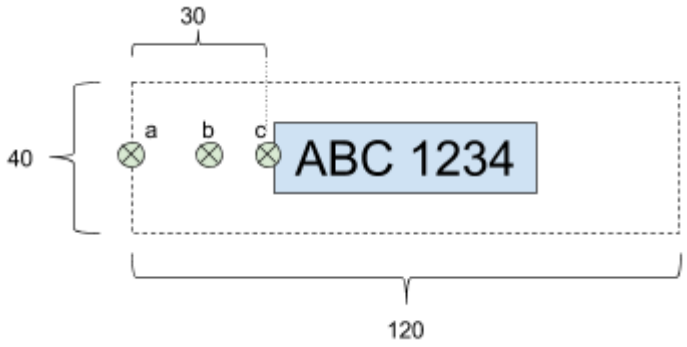

Figura 26: Região onde amostras são coletadas. Esta região delimita os centros das partições, então *pixels* fora dela podem ser coletados. Partições centradas dentro da região delimitada são coletados de 4 em 4 *pixels*, sendo que cada uma dessas regiões é deslocada 2 *pixels* para a direita ou para a esquerda. Em verde estão marcados três pontos, a, b e c, que serão atribuídos respectivamente aos rótulos 0, $^1/_2$ e 1 por estarem respectivamente a uma distância normalizada de $^1/_2$, $^3/_8$ (meio do caminho) e $^1/_4$ (autoria própria).

começa a decair de 1 para 0 (que são codificados como 255 e 0). O ponto (a) é onde a função atinge o valor 0, e o ponto (b) está precisamente no meio do caminho, possuindo o valor 0,5 (codificado como 128).

Após coletar os exemplos de placas contidas na região acima indicada ocorre a coleta de imagens fora da região, denominada coleta de "exemplos negativos". A todos os exemplos coletados dessa maneira é atribuído o rótulo 0.

O algoritmo que coleta esses exemplos passou por vários aprimoramentos. A versão final, que vai ser descrita, melhorou consideravelmente o desempenho do treinamento comparado com as versões iniciais.

A primeira, e mais importante otimização foi a eliminação de exemplos parecidos. Quando são coletados exemplos de regiões da imagem que não são placas de carro pode-se acabar obtendo um recorte que inclui apenas o asfalto. Se outro recorte for feito em outro *frame* na mesma região pode acabar sendo um exemplo muito parecido, como ilustrado na figura 27.

Por este motivo o *software* mantém os últimos 100 *frames* processados. Quando um "exemplo negativo" vai ser coletado em uma região a imagem é comparada com os *pixels* da mesma região nestes *frames* que foram armazenados. O critério de comparação é:

$$d_C = \frac{1}{N}\sum_{n=1}^{N}\sqrt{|R_C[n] - I_C[n]|} \tag{24}$$

Onde $R_C[n]$ é o valor do canal de cor "C" do n-ésimo pixel da partição de referência, e $I_C[n]$ é o mesmo canal do n-ésimo *pixel* da partição que está sendo testada. Quando o valor

Figura 27: Exemplo de duas regiões iguais em *frames* diferentes. As regiões possuem as mesmas coordenadas, e contém *pixels* de fundo da imagem (autoria própria).

da média dessa métrica para os três canais é menor que $\sqrt{8}$ as imagens são consideradas semelhantes, e a amostra é recusada. Foram testadas métricas como média do módulo da diferença, e média da diferença quadrática, mas tiveram desempenho inferior ao serem comparados usando a percepção humana como referência. A figura 25 mostra o efeito desta otimização. Cada retângulo azul mostra a região onde um exemplo negativo foi coletado. Observa-se que o asfalto não está sendo coletado.

A segunda otimização foi a distância entre duas amostras negativas. Duas amostras muito próximas têm pouco valor para treinamento da rede neural devido à invariância a deslocamento. Por isso, para uma amostra ser aceita como "exemplo negativo" ela precisa estar a mais de 4 *pixels* de todos os outros exemplos coletados.

A terceira otimização foi o processo de escolha das coordenadas onde os exemplos são coletados. Inicialmente usava-se um grid regular, mas essa abordagem acabava coletando uma quantidade muito grande de exemplos negativos.

Como isso a rede neural acabava aprendendo a favorecer valores negativos, pois errar "para mais" acabava sendo punido mais severamente que errar "para menos" durante a otimização. Para poder balancear a quantidade de exemplos negativos e positivos foi necessário substituir o grid linear, já que ele favorecia coletar exemplos negativos na parte superior da imagem, pois a coleta terminava antes da imagem chegar na parte inferior.

Para resolver isso foi adotado um processo que escolhe coordenadas usando uma distribuição uniforme e testa se nas coordenadas existe um exemplo negativo válido, repetindo este processo até encontrar o número desejado de amostras. Infelizmente este algoritmo é $O(\infty)$ no pior caso. Por isso um novo algoritmo de busca que tenha probabilidade próxima de uniforme, encontre todas as soluções, e sempre termine precisou ser criado. Este algoritmo é uma modificação do BFS (*breadth-first search*), no qual a ordem na qual a busca é adicionada à pilha é

Tabela 3: Exemplos de treinamento gerados para cada vídeo

Entrada	Saída	Tamanho	Registros Gerados
video1.avi	video1.nn1.bin	1,1 GiB	64.714
video2.avi	video2.nn1.bin	901 MiB	89.672
TOTAL		2,07 GiB	154.386

aleatória. O algoritmo pega de uma pilha uma região retangular na qual deve encontrar pontos válidos e usa um gerador aleatório uniforme para escolher coordenadas e testar. Se o teste falhar divide a região em duas sub-regiões na direção onde a imagem for maior (altura ou largura) e adiciona as regiões na pilha em ordem aleatória. Cada vez que coordenadas válidas são encontradas elas são adicionadas na lista de resposta. Se a lista possui o número solicitado de respostas o algoritmo termina, caso contrário continua.

A tabela 3 mostra os dados coletados quando o software foi executado com todas as otimziações referidas.

Um *script* em *Bash* foi escrito para partir um arquivo destes em arquivos menores, contendo aproximadamente 128 amostras cada. Isso gerou 699 arquivos a partir de *video1.nn1.bin* e 504 arquivos a partir de *video2.nn1.bin*.

Finalmente, um diretório foi criado contendo links simbólicos para todos os 1203 arquivos. Este diretório é o resultado final da geração de exemplos, e contém todas as amostras coletadas de todos os vídeos.

5.5.3 TREINADOR

O treinador é um módulo de *software* onde se configura uma topologia de rede neural convolucional, que é treinada usando os dados produzidos pelo gerador de dados de treinamento. Quando o software é executado informações de *log* são coletadas, e um utilitário chamado *TensorBoard* pode ser executado neste diretório para mostrar o progresso do treinamento. O utilitário é um servidor *web*, e a página correspondente é mostrada na figura 28. Para realizar o treinamento da rede neural o grafo apresentado na figura 29 foi construído no *TensorFlow*.

O bloco “FileRead” possui uma lista com todos os arquivos de treinamento gerados pelo gerador de exemplos de treinamento. Essa lista é embaralhada, e então cada um dos registros é lido em sequência. Cada imagem é representada por um tensor de inteiros de 8 bits com

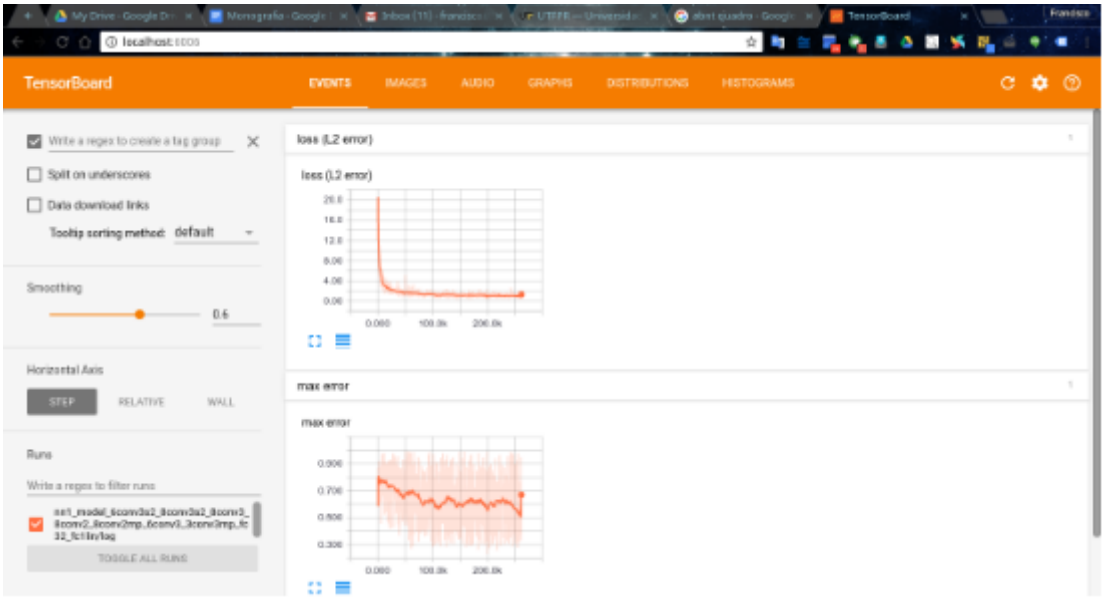

Figura 28: Página do *TensorBoard* mostrando progresso do treinamento. Esta é uma página *web* que permite observar a evolução do treinamento (autoria própria).

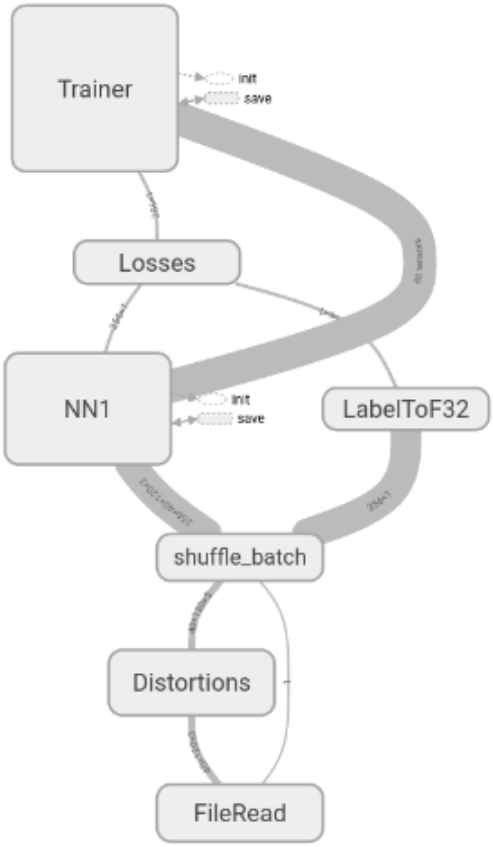

Figura 29: Grafo usado para treinamento da rede neural convolucional. Esta imagem mostra o grafo do *TensorFlow* usado para treinar a rede neural, designada "NN1" (autoria própria).

dimensões $40 \times 120 \times 3$, e o rótulo da imagem é representado com um tensor unidimensional com dimensão 1.

Como a rede neural está sendo treinada usando apenas dois vídeos existe o risco da rede neural treinada ficar muito sensível a fatores como a iluminação. Para impedir que isso aconteça as imagens são distorcidas durante o treinamento no bloco de distorções, que está ilustrado na figura 30. A imagem é convertida para ponto-flutuante com canais no intervalo $[0; 1]$, conforme requerido pelas funções de processamento de imagem do *TensorFlow*. No final das distorções a imagem é convertida novamente para inteiro de 8 bits por canal. O motivo para isso é fazer com o que o bloco "NN1", que implementa a rede neural, tenha a sua entrada em um formato que permite ser alimentada diretamente a partir das partições da imagem, sem nenhuma conversão.

Figura 30: *Pipeline* de distorção de imagens. A imagem é convertida para ponto-flutuante com cada canal no intervalo $[0; 1]$, é adicionado ruído normal e a saturação, brilho e contraste são alterados conforme uma distribuição uniforme. A imagem é então convertida novamente para inteiro de 8 bits por canal (autoria própria).

O bloco "*Noise*" adiciona ruído normal à imagem somando-a com um tensor com as mesmas dimensões onde o valor de cada canal é gerado por uma distribuição normal de média 0 e desvio padrão 0,06. O bloco "*Saturation*" altera a saturação entre 0,2 a 1,5, sendo que 1 representa manter a saturação inalterada. O bloco "*Brightness*" faz uma alteração no brilho aumentando ou diminuindo em até 40%. O bloco "*Contrast*" faz uma alteração de contraste entre 0,6 e 1,4, sendo que 1 representa manter o brilho. As escolha do valor em todos os casos é aleatória de acordo com distribuição uniforme.

O resultado das distorções, juntamente com o rótulo da imagem são fornecidos para o

bloco "*shuffle_batch*". Este bloco tem a função de embaralhar os dados e agrupar imagens em lotes. Este bloco usa 8 *threads* para consumir os dados dos blocos anteriores, o que significa que leitura e embaralhamento ocorrerão em paralelo. Quando 2560 imagens são lidas, elas são embaralhadas, e elas são agrupadas em lotes de 256 imagens, gerando tensores com dimensões $256 \times 40 \times 120 \times 3$. Como cada arquivo contêm 128 registros, então as imagens sendo lidas contém, no pior caso, uma amostra aleatória proveniente de 20 arquivos. Os arquivos em si são lidos em ordem aleatória, o que gera uma amostragem razoavelmente diversa das informações de entrada. Quanto maior o tamanho do lote maior a precisão da estimativa de erro. A escolha por um lote de 256 imagens foi o maior suportado pela GPU usada para treinamento (GTX 750M com 2 GiB de RAM). Ao treinar usando CPU valores ainda maiores foram usados.

Se o treinamento for rodado por tempo suficiente, todas as imagens serão lidas. Se o treinador precisar de mais exemplos, as mesmas imagens serão lidas novamente, em ordem diferente. Porém, graças ao sistema de distorção de imagens que é parte do *pipeline* de leitura, uma mesma imagem passará novamente pelo *pipeline* de distorções, efetivamente gerando uma imagem diferente.

Os tensores que saem do embaralhador (bloco "*shuffle_batch*") são fornecidos à rede neural, que está no bloco "NN1", que está ilustrado na figura 31.

Pode-se ver pela espessura das linhas que conectam os blocos que a quantidade de informação reduz ligeiramente em cada etapa. Como a rede neural precisa de um tempo de propagação baixo, por estar operando em vídeo, algumas técnicas foram usadas para reduzir o número de computações.

A primeira estratégia é a redução agressiva do tamanho dos tensores nas duas primeiras camadas, inspirado em (SZEGEDY et al., 2015a). *Stride* é usado para fazer um reamostragem da imagem. Este método usa mais CPU do que reamostragem por métodos como interpolação bilinear, porém permite acesso à textura da imagem original. A primeira camada usa um *kernel* de 2×2 ao invés de 3×3, reduzindo o número de multiplicações de 9 para 4.

Não está sendo usada nenhuma convolução maior que 3×3. Szegedy et al. (2015a) defende que, pelo menos em alguns casos, convoluções 5×5 podem ser substituídas por duas convolução sucessivas 3×3, reduzindo o número de multiplicações de 25 para $2 \cdot 9 = 18$.

Nas últimas camadas está sendo usado *maxpool* 2×2. Esta estratégia é computacionalmente mais cara que usar convolução com *stride* maior que 1, pois quatro valores são calculados e 3 deles são descartados. Como as imagens são menores no final da rede neural, optou-se pela operação mais cara. O efeito em tempo de propagação é negligenciável, o efeito na qualidade

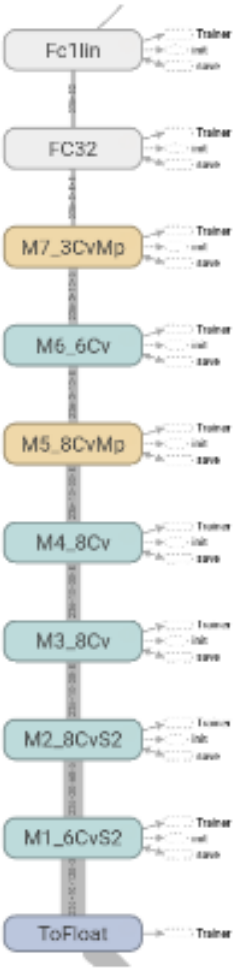

Figura 31: *Pipeline* descrevendo a rede neural convolucional profunda (autoria própria).

da detecção não foi medido.

Outro ponto importante é o controle do número de graus de liberdade. Os primeiros modelos tinham mais de 50.000 graus de liberdade, e este valor foi reduzido para 5.242. Menos parâmetros implica várias vantagens, como treinamento mais rápido, menos uso de memória, capacidade de usar lotes maiores e menor chance de *overfitting*.

O esforço de melhorar o desempenho da rede neural passou a ser consideravelmente mais produtivo quando um processo foi adotado, no qual somente eram testadas topologias que mantinham ou reduziam o número de parâmetros totais. Quando esta estratégia foi adotada a produtividade no que diz respeito a otimização destes parâmetros teve um salto considerável.

Na tabela 4 é apresentada a topologia da rede neural convolucional final. Ela possui 7 camadas convolucionais com parâmetros (*maxpol* e ReLu não contadas) e duas camadas totalmente conectadas. A primeira camada (M1) é uma convolução 2×2 aplicada com stride 2×2. Isso significa que todos os pixels são visitados uma vêz por cada filtro desta camada, e tem com

Tabela 4: Camadas da Rede Neural Implementada

Nome	Tensor de Entrada	Operação	DOF
M1	$256 \times 40 \times 120 \times 3$	6 cv2x2 s2x2 relu	78
M2	$256 \times 20 \times 60 \times 6$	8 cv3x3 s2x2 relu	440
M3	$256 \times 10 \times 30 \times 8$	8 cv3x3 relu	584
M4	$256 \times 10 \times 30 \times 8$	8 cv3x3 relu	584
M5	$256 \times 10 \times 30 \times 8$	8 cv3x3 relu mp2x2	584
M6	$256 \times 5 \times 15 \times 8$	6 cv3x3 relu	438
M7	$256 \times 5 \times 15 \times 6$	3 cv3x3 relu mp2x2	165
-	$256 \times 3 \times 8 \times 3$	flatten	0
FC32	256×72	fc32 relu	2.336
FC1	256×32	fc1 linear	33
SAÍDA	256		
TOTAL			5.242

objetivo fazer uma subamostragem sem consumir muito tempo de processamento. A camada M2 também usa stride 2×2, mas tem filtros 3×3, que são mais usuais. A partir da camada M3 o stride é sempre 1×1, não havendo mais subamostragem. Nas camadas M5 e M7 é feito *maxpool*. Depois de M7 o tensor é planificado para poder ser usado pelas duas camadas convolucionais, que possuem respectivamente 32 e 1 neurônio. O número de neurônios na camada FC32 possui quase metade dos parâmetros da rede neural, e afeta bastante o desempenho de classificação e o tempo de processamento. Este valor foi escolhido de forma a balancear estes fatores.

A saída da rede neural, denominada "A", é um tensor unidimensional com dimensão 256, contendo um escalar que representa o valor estimado da função para uma imagem. Para o treinamento, a saída da rede neural vai ser comparada com os rótulos que estavam nos dados de treinamento, representados pelo tensor "B", e também forma um tensor unidimensional com dimensão 256. O erro entre as duas medidas é mapeado para um único escalar, denominado perda, ou *loss*, pela metade da norma L2 entre os dois tensores sem a raiz quadrada:

$$loss = \frac{1}{2} \sum_{n=1}^{N} (A[n] - B[n])^2 \tag{25}$$

Esta medida foi usada porque é calculada eficientemente pelo *TensorFlow* através da função **tf.nn.l2_loss(t, name=None)**. O *software* de treinamento usa um otimizador Adam (KINGMA; BA, 2014) para atualizar os parâmetros treináveis da rede neural de forma a minimizar este valor.

As figuras 32 e 33 mostram a evolução do treinamento durante uma sessão de 25 horas treinadas em um notebook Core i5 usando GPU. Durante este tempo foram consumidos mais de 66 milhões de imagens, o que consumiu todos os 2 GiB de dados e seus 154.386 exemplos de imagens cerca de 428 vezes.

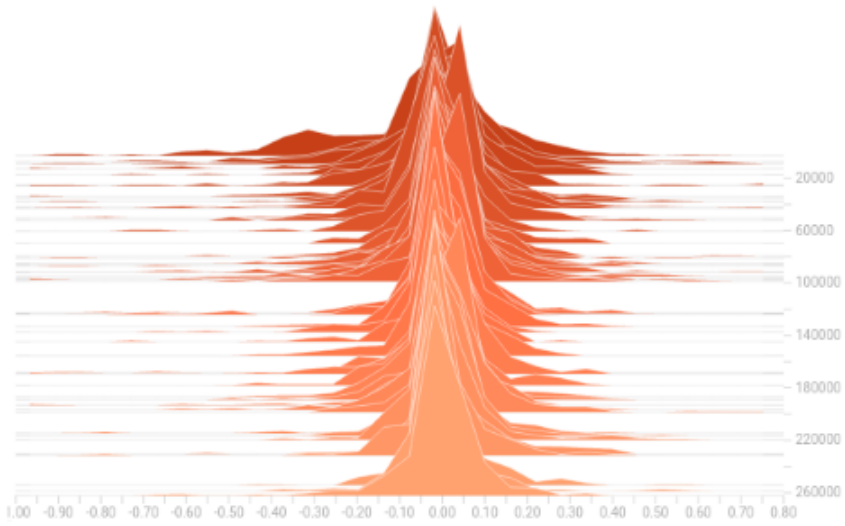

Figura 32: Histograma temporal do erro classificação. Podem-se observar os erros se concentrando cada vez mais próximo de 0 à medida que o treinamento avança. A escala representa lotes de 256 imagens cada, de forma que está representado o processamento de 66 milhões de imagens (autoria própria).

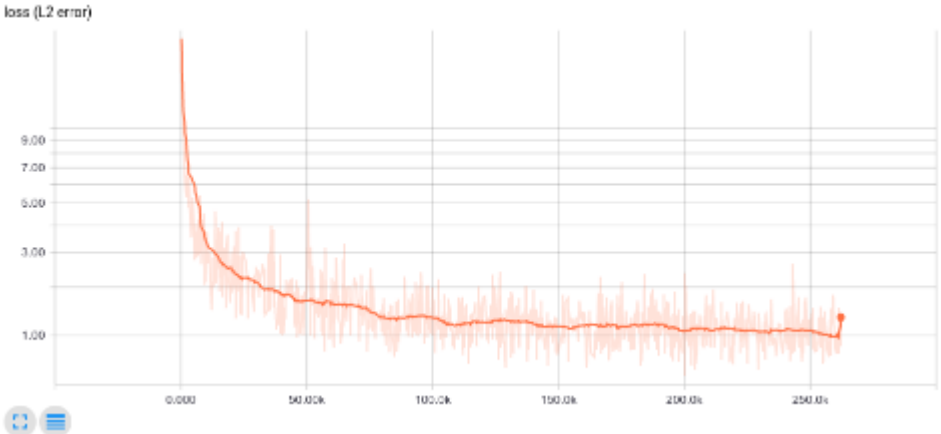

Figura 33: Gráfico do erro de treinamento. Evolução da função de perda que o otimizador está minimizando, amortecido usando método média local. A abscissa é o número do lotes, e cada lote possui 256 imagens (autoria própria).

5.5.4 PLAYER

Figura 34: Interface com o usuário do módulo *Player*. No momento que a tela foi capturada o *software* está exibindo um vídeo enquanto destaca as placas que está localizando (autoria própria).

O quarto módulo de *software* usa a rede neural treinada para reproduzir o conteúdo de um vídeo ou da câmera enquanto destaca as placas de veículos localizadas. A figura 34 mostra a interface do usuário no momento em que está exibindo um vídeo quando três veículos estão passando. As placas veiculares estão sendo destacadas.

O *software* implementa o método de classificação proposto nesta monografia por completo. O método usado para gerar as coordenadas das placas a partir de todos os escores da imagem (mapeamento) é bastante simples: qualquer partição que possua valor maior que 0,8 (ver seção 5.6.1) e seja maior que que o valor das partições vizinhas, incluindo diagonais, deve estar mais próxima do centro de uma placa. O algoritmo vai assumir que a placa está no centro desta partição. Como a placa pode de fato estar a uma distância de $P_x/2 = 60$ *pixels* para a direita ou para a esquerda, e $P_y/2 = 20$ *pixels* para cima ou para baixo.

Não foi implementado nenhum método para usar a correlação temporal das informações coletadas ou rejeição de dados espúrios.

Para o *video1.avi* a rede neural é executada 253 vezes para cada *frame* do vídeo, sendo que uma linha completa contendo 11 partições é fornecida para a rede neural por vez.

O *software* é capaz de operar em qualquer vídeo que possa ser lido pelo *OpenCV3* e pode usar a imagem da câmera do computador.

5.6 EXPERIMENTOS E DESEMPENHO

Nesta seção serão apresentados os experimentos realizados com a implementação do *software* e os seus resultados.

5.6.1 DESEMPENHO DE LOCALIZAÇÃO

Como o método de localização proposto é aproximado, a abordagem para teste de desempenho é baseada na contagem de placas corretamente ou incorretamente localizadas. Para que uma placa seja considerada corretamente localizada, basta que ela esteja a menos de 60 *pixels* de distância na largura e menos de 20 *pixels* na altura.

Para a medição de desempenho serão usadas as métricas *precision* (equação 26) e *recall* (equação 27).

$$\text{Precision} = \frac{vp}{vp + fp} \qquad (26) \qquad\qquad \text{Recall} = \frac{vp}{vp + fn} \qquad (27)$$

Onde vp é o número de verdadeiros positivos, fp é o número de falsos positivos e fn é o número de falsos negativos. Estes dados serão obtidos da seguinte maneira:

- Quando o *software* identificar uma placa em um ponto do *frame* e de fato houver uma placa entre as manualmente marcadas, tal que a distância entre os centros das duas seja inferior ao erro, será contabilizado um *verdadeiro positivo*, ou *vp* (figura 35, placa “a”). A placa é removida da lista de placas manualmente marcadas para não ser usada novamente, pois já foi localizada (figura 35, placa “b”).;
- Se no caso anterior não houver placa manualmente marcada tal que o erro seja inferior ao máximo será contabilizado um falso positivo, ou *fp* (figura 35, placa “c”);
- as placas que foram manualmente marcadas e não forem localizadas são contabilizadas como falso negativos, ou *fn* (figura 35, placa “A”).

Para usar corretamente o método proposto é preciso escolher um limiar de classificação. Se o escore de uma partição for maior que este limiar considera-se que a partição possui uma placa veicular. Como este escore afeta as métricas *precision* e *recall*, foi feito um levantamento da *curva precision-recall* para determinar o valor mais apropriado de limiar a ser usado.

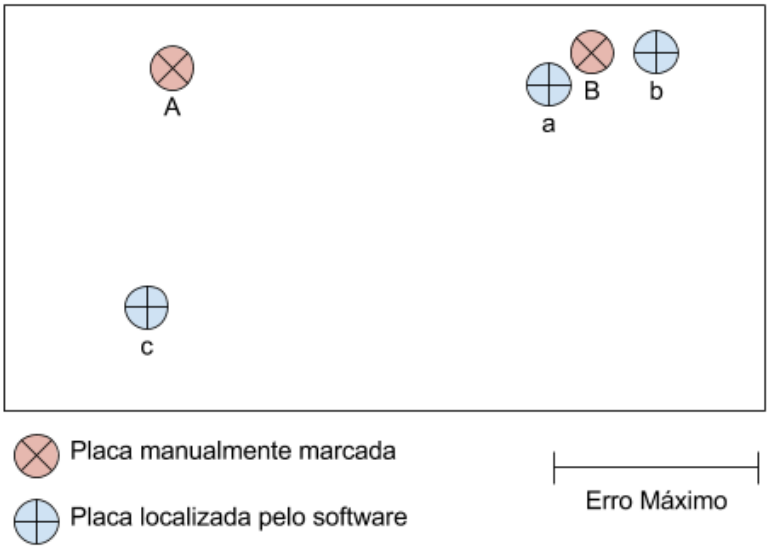

Figura 35: Contagem de acertos e erros de localização. Cada círculo representa o centro de uma placa. Constam 1 *verdadeiro positivo* (B/a), 2 falsos positivos (b, c), e um *falso negativo* (A). Próximo a placa "B" foram localizadas duas placas, mas só há uma, então uma delas é contada como *falso positivo* (autoria própria).

A tabela 5 e o gráfico na figura 37 mostram esta curva para os dados de teste. Apenas para referência, o gráfico também foi calculado para o conjunto de treinamentos, por ser maior, gerando o gráfico da figura 36.

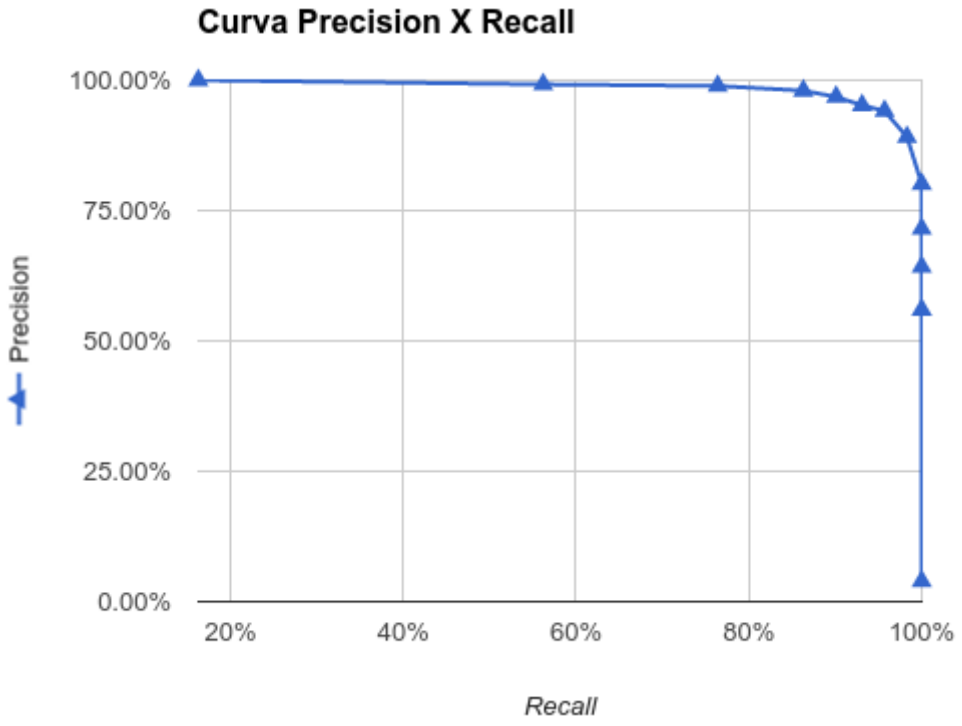

Figura 36: Gráfico da curva *Precision-Recall* (amostras de treinamento). Comparada com a mesma curva para os dados de teste esta é mais suave. Isso se deve maior quantidade de dados no conjunto de treinamento, o que gera menos incerteza (autoria própria).

Tabela 5: Erros e acertos de localização

Limiar	Recall	Precision
0	100,0%	3,44%
0,1	100,0%	53,8%
0,14	100,0%	57,9%
0,2	100,0%	67,2%
0,3	100,0%	78,0%
0,5	100,0%	94.8%
0,7	100,0%	94,8%
0,8	**96,7%**	**98,9%**
0,9	94,6%	98,9%
0,93	91,3%	98,8%
0,97	80,4%	98,7%
0,99	65,2%	98,4%
1,01	19,6%	100,0%

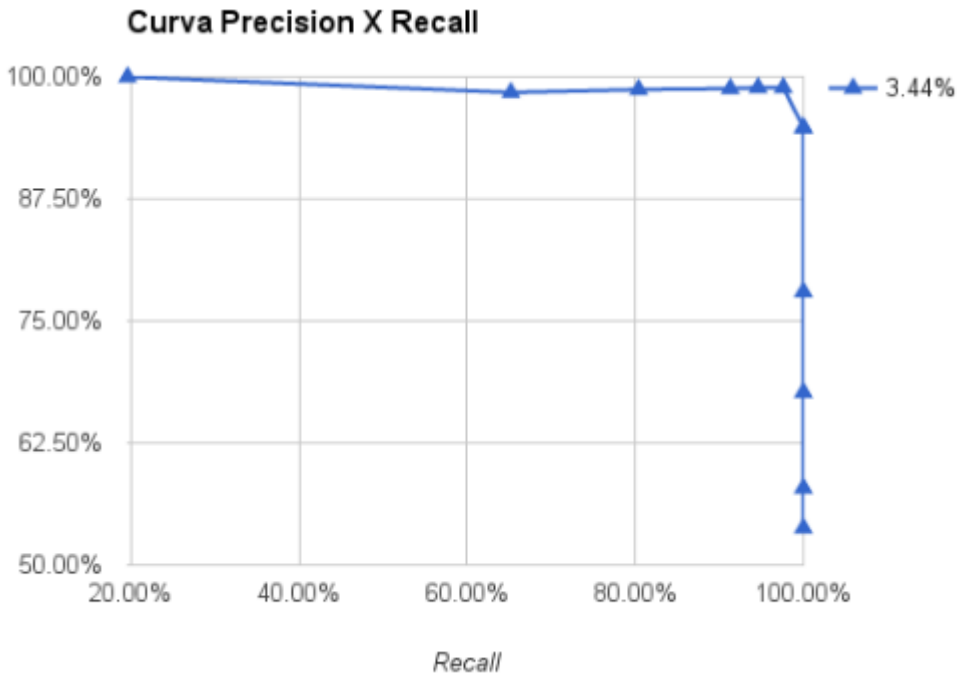

Figura 37: Gráfico da curva *Precision-Recall* (amostras de teste). Pode-se observar como o limiar de classificação afeta *precision* e *recall* (autoria própria).

Tendo sido escolhido o limiar 0.8 para aceitar uma placa, as métricas *precision* e *recall* foram calculados para todos os conjuntos de dados marcados disponíveis, tanto de teste quanto de treinamento. Os resultados obtidos estão apresentados na tabela 6. Os indicadores de desempenho foram melhores no conjunto de dados de teste do que no conjunto de treinamento. Isso se deve ao fato de que a implementação, como foi feita, não detecta placas que estejam próximas a borda direita e inferior da imagem. O conjunto de testes foi preparado tomando cuidado para não usar *frames* onde veículos se encontrem nesta região.

O resultado de *precision* no video2 foi apenas 59,0%, causado pela quantidade excessiva de falso positivos. Observou-se que uma única das partes da imagem que pertence ao asfalto, foi marcada em aproximadamente 50% dos *frames* como placa, gerando falso positivos. Atribui-se este problema a resolução deste vídeo, que é 1080×1920, e portanto requer uma grande quantidade de marcações para que seja coletada uma quantidade suficiente de amostras de "não-placa" para eliminar este problema, já que o *software* de coleta de exemplos de treinamento tenta balancear a quantidade de exemplos positivos e negativos.

5.6.2 FUNÇÃO

O método proposto envolve fazer a regressão de uma função. Esta seção tem com objetivo medir o desempenho da rede neural nesta tarefa. As figuras 38 e 39 mostram 3 carros,

Tabela 6: Erros e acertos de localização

Vídeo	VP	FP	FN	Precision	Recall
video1.avi (teste)	**89**	**1**	**3**	**98,9%**	**96,7%**
video1.avi (treinamento)	217	11	16	95,2%	93,1%
video2.avi (treinamento)	92	64	4	59,0%	95,8%

Tabela 7: Taxa de *Frames* por Segundo

Vídeo	Resolução	FPS (GPU)	FPS (CPU)	Melhoria
camera	480×640	15,8	9.12	73%
video1.avi	480×768	14,8	8,08	84%
video2.avi	1080×1920	3,29	1,51	117%
video3.avi	1080×1920	3,34	1,51	121%

uma moto e um ônibus, sendo que o classificador foi aplicado *pixel*-a-*pixel* na região próxima da placa, e o seu valor representado em um gráfico 3D. Este gráfico pode ser comparado com o gráfico mostrado na figura 20, que mostra o gráfico ideal.

Estes dados foram todos levantados usando o vídeo3.avi, que não foi usado para treinamento. Os cinco *frames* foram escolhidos ao acaso, garantindo-se apenas que as cinco imagens sejam suficientemente diferentes umas das outras.

A capacidade de produzir estes gráficos foi incluída no *software* "*player*", e pode ser invocada para qualquer *frame* de qualquer vídeo.

A figura 40 mostra a rede neural sendo aplicada no *frame* 11.197 do video1. Este *frame* não faz parte do conjunto de treinamento. Pode-se verificar claramente onde estão as duas placas veiculares.

5.6.3 TEMPO DE CLASSIFICAÇÃO

Para os testes de tempo de classificação foi usado o módulo *player*. Este módulo mostra uma janela com o vídeo sendo classificado enquanto envia para *stdout* uma série de informações de desempenho. Estes dados foram usados nas medições relacionadas a tempo de classificação.

Observa-se na tabela 7 que o *player* não consegue atingir a taxa de 25 *frames* por

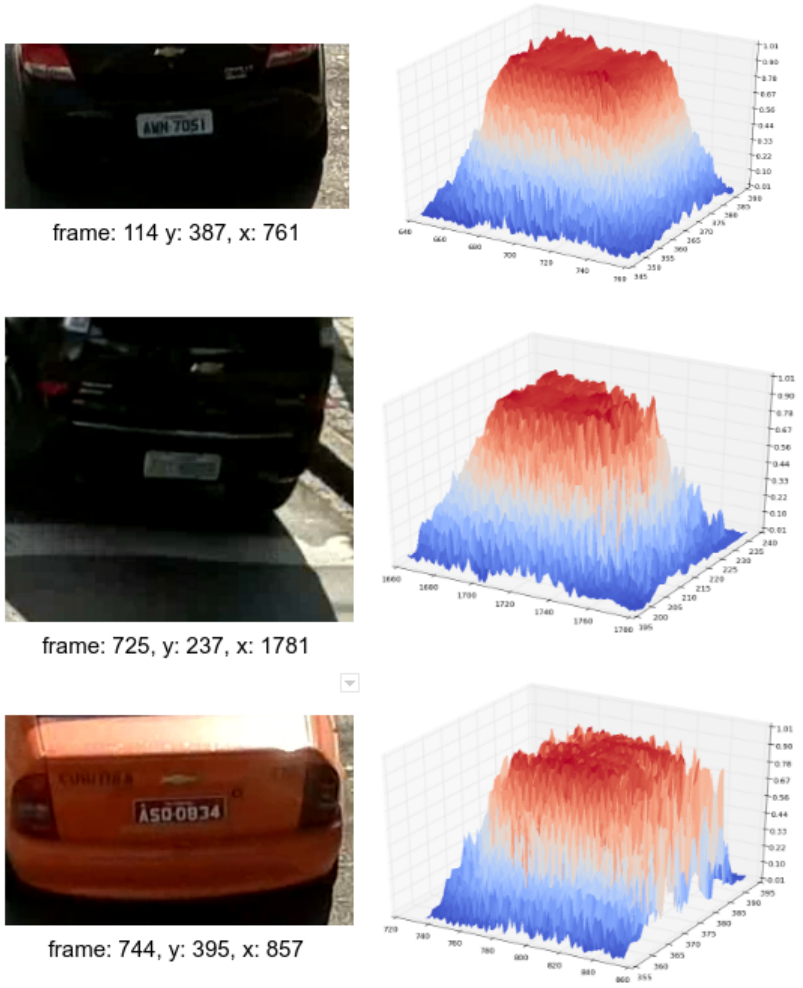

Figura 38: Gráfico 3D da função aplicada a carros. São destacados três carros, um com boa textura, placa razoavelmente visível, outra com imagem com perda resultante de compactação, e um terceiro com orientação e cores diferentes, inclusive usando letras mais claras que o *frame* da placa. O gráfico mostra o resultado do classificador quando ele está centrado em cada um dos *pixels* próximo à placa. As abscissas referem-se as coordenadas do canto superior-esquerdo da partição, não ao seu centro (autoria própria).

segundo de nenhum dos três vídeos, mesmo com uso de GPU. A taxa de *frames* aumenta consideravelmente quando a GPU é usada. Entre os casos testados o ganho aumenta quando a resolução do vídeo aumenta. A figura 41 mostra o resultado em forma de gráfico.

O *player* envia para o *stdout* o tempo gasto em cada uma das tarefas durante o processamento do *frame*. Coletando-se estes dados e calculando-se as médias do tempo gasto em cada tarefa foi gerado o gráfico na figura 42. Pode-se ver claramente que a vasta maioria do tempo é gasta executando os grafos do *TensorFlow*. Também foi possível confirmar que o ganho do desempenho no caso do uso do GPU foi consequência do *TensorFlow* consumir menos tempo para executar. Observou-se que na implementação CPU o *TensorFlow* tomou 94,9% do tempo de processamento do *frame*, e no caso que usa GPU tomou 88,1%.

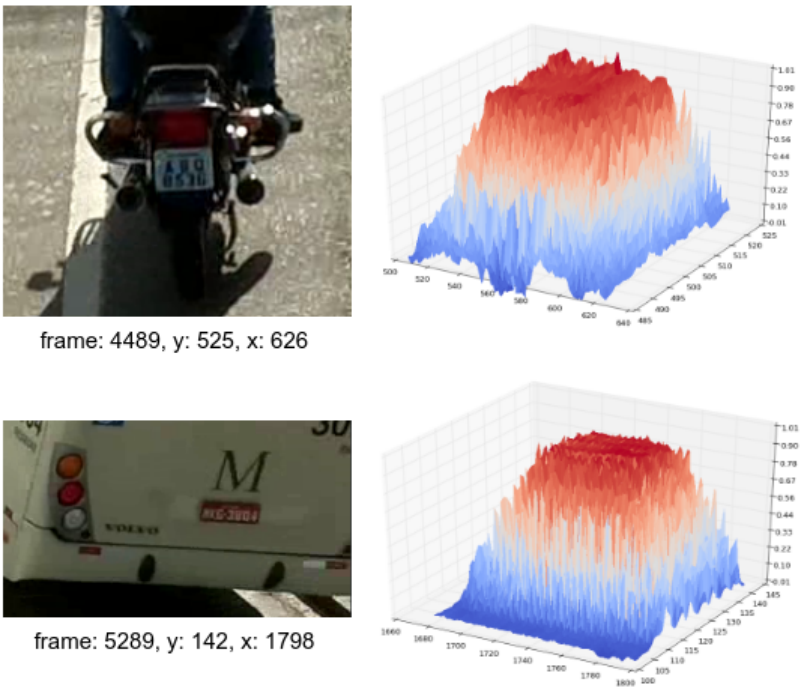

Figura 39: Gráfico 3D da função aplicada a moto e ônibus. Observa-se uma moto e um veículo e o gráfico resultante de aplicar o classificador em torno das placas. As abscissas referem-se as coordenadas do canto superior-esquerdo da partição, não ao seu centro (autoria própria).

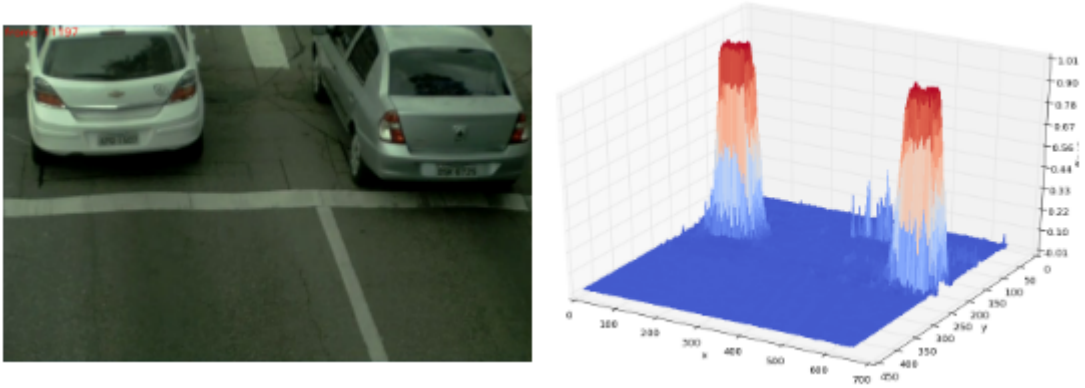

Figura 40: Gráfico 3D mostrando a função aplicada à um *frame* inteiro. Pode-se ver claramente onde as placas dos dois veículos se encontram. (autoria própria).

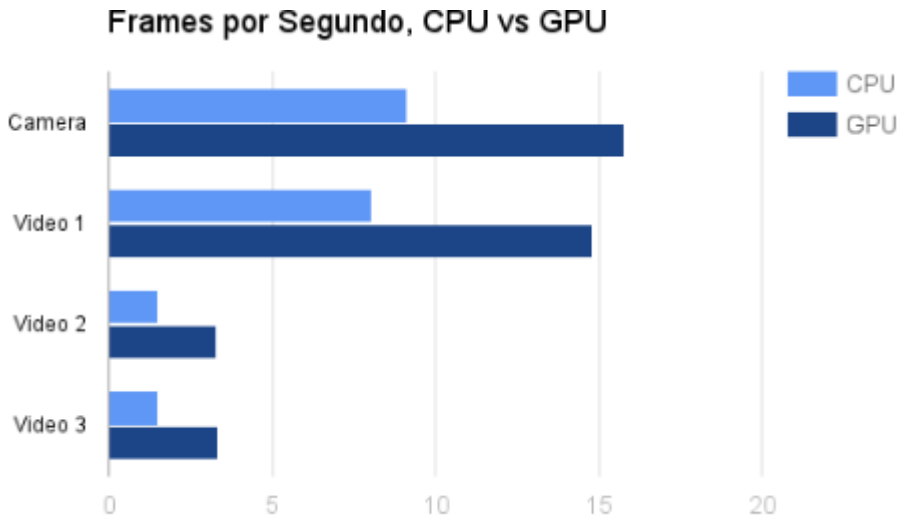

Figura 41: Taxa de *frames* por segundo no módulo *Player*. Gráfico mostrando desempenho em *frames* por segundo para localizar placas na câmera VGA do computador, e nos três vídeos. Computador é um Core i5 com placa de vídeo GTX 750M. Maior é melhor (autoria própria).

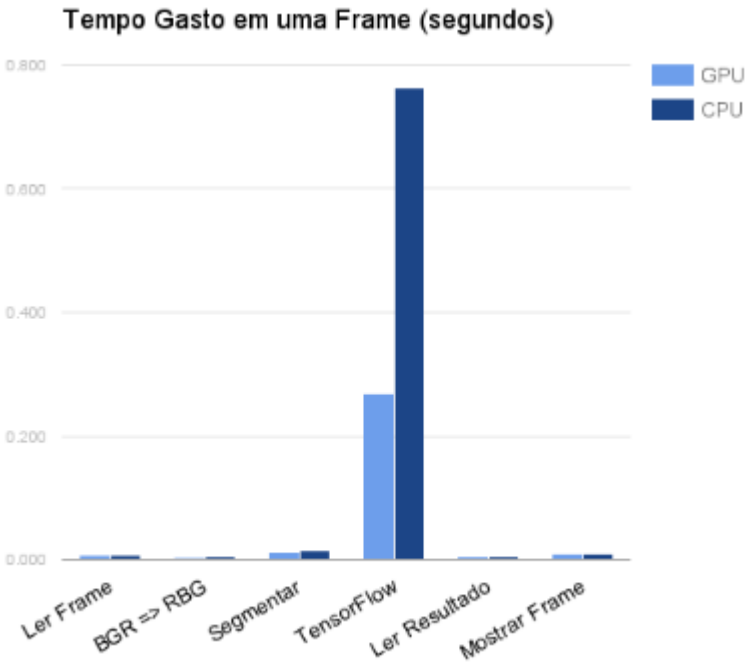

Figura 42: Tempo gasto em cada subtarefa no processamento de um *frame*. Gráfico mostra que, ao processar um vídeo 1080×1920 a grande maioria do tempo é gasto executando o grafo do *TensorFlow*. Outras tarefas executadas pelo código em Python consomem um percentual pequeno do tempo do *frame*. Computador é um Core i5 com placa de vídeo GTX 750M (autoria própria).

6 CONCLUSÕES

Os resultados obtidos mostram que o método proposto é capaz de localizar placas veiculares com excelentes valores de *precision* e *recall*, enquanto executa em poucos milissegundos. A rede neural reproduz corretamente o valor da função que está modelando, estimando a posição da placa veicular na sua entrada, apesar de ter sido treinada com substancial adição de ruído e outras distorções, demonstrando a validade do método.

O método de particionamento usado para determinar onde a rede neural é aplicada, não foi suficiente para permitir atingir taxa de *frames* nominal de nenhum dos vídeos usados no computador testado. O aumento dessa taxa pode ser conseguido de várias formas, incluindo o uso de uma GPU mais rápida, otimização da topologia da rede neural, melhor agendamento de lotes e melhor distribuição de tarefas para CPU e GPU.

Uma consequência negativa do método de particionamento usado é a imprecisão na posição da placa. O algoritmo usado para converter os escores para coordenadas usou "máxima local", e por isso, quando uma placa é localizada, a posição real dela pode estar $\pm 20 \times \pm 60$ *pixel*, o que pode ser excessivo para certas aplicações. No entanto é possível reduzir pela metade com um método mais sofisticado de conversão dos escores em coordenadas usando os quatro maiores valores. Se houver *budget* computacional é possível também aproximar mais as partições, reduzindo arbitrariamente estes valores.

Para futuros trabalhos sugere-se adicionar ao modelo aqui proposto a capacidade de fazer segmentação e OCR, de forma a produzir um sistema completo de identificação de placas veiculares. Dois aspectos do método proposto que podem ser melhorados são a função modelada pela rede neural, que é $C0$ e poderia ser $C1$ para melhorar a regressão, e o processo de cálculo de coordenadas das placas a partir dos escores da rede neural que poderia aumentar a precisão da localização. Entre os aspectos práticos, e de implementação, sugere-se continuar o trabalho de otimização da rede neural para reduzir a quantidade de computações sem gerar muito impacto nos resultados de classificação.

REFERÊNCIAS

ANAGNOSTOPOULOS, C.-N. E. et al. License plate recognition from still images and video sequences: A survey. **IEEE Transactions on intelligent transportation systems**, IEEE, v. 9, n. 3, p. 377–391, 2008.

BOSER, B. E.; GUYON, I. M.; VAPNIK, V. N. A training algorithm for optimal margin classifiers. In: ACM. **Proceedings of the fifth annual workshop on Computational learning theory**. [S.l.], 1992. p. 144–152.

CORTES, C.; VAPNIK, V. Support-vector networks. **Machine learning**, Springer, v. 20, n. 3, p. 273–297, 1995.

DALAL, N.; TRIGGS, B. Histograms of oriented gradients for human detection. In: IEEE. **2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)**. [S.l.], 2005. v. 1, p. 886–893.

GILBERT, N.; TERNA, P. How to build and use agent-based models in social science. **Mind & Society**, Springer, v. 1, n. 1, p. 57–72, 2000.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing (3rd Edition)**. 3. ed. Pearson, 2007. ISBN 9780131687288. Disponível em: <http://amazon.com/o/ASIN/013168728X/>.

GUO, F.; KRISHNAN, R.; POLAK, J. Short-term traffic prediction under normal and incident conditions using singular spectrum analysis and the k-nearest neighbour method. In: IET. **Road Transport Information and Control (RTIC 2012), IET and ITS Conference on**. [S.l.], 2012. p. 1–6.

HASANPOUR, S. H. et al. Lets keep it simple: using simple architectures to outperform deeper architectures. **arXiv preprint arXiv:1608.06037**, 2016.

HAWKINS, D. M. The problem of overfitting. **Journal of chemical information and computer sciences**, ACS Publications, v. 44, n. 1, p. 1–12, 2004.

HE, K. et al. Deep residual learning for image recognition. **arXiv preprint arXiv:1512.03385**, 2015.

HUBEL, D. H.; WIESEL, T. N. Receptive fields and functional architecture of monkey striate cortex. **The Journal of physiology**, Wiley Online Library, v. 195, n. 1, p. 215–243, 1968.

JAIN, A. S.; KUNDARGI, J. M. Automatic number plate recognition using artificial neural network. **International Research Journal of Engineering and Technology (IRJET)**, Irjet, v. 2, n. 4, p. 1072–1078, 2015.

KIM, K. K. et al. Learning-based approach for license plate recognition. In: IEEE. **Neural Networks for Signal Processing X, 2000. Proceedings of the 2000 IEEE Signal Processing Society Workshop**. [S.l.], 2000. v. 2, p. 614–623.

KINGMA, D.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

KRANTHI, S.; PRANATHI, K.; SRISAILA, A. Automatic number plate recognition. **Int. J. Adv. Tech**, v. 2, n. 3, p. 408–422, 2011.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LECUN, Y. et al. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, IEEE, v. 86, n. 11, p. 2278–2324, 1998.

LIN, Y. et al. Imagenet classification: fast descriptor coding and large-scale svm training. **Large scale visual recognition challenge**, 2010. Acessado: 2016-11-20. Disponível em: <http://www.image-net.org/challenges/LSVRC/2010/ILSVRC2010_NEC-UIUC.pdf>.

LUVIZON, D. C.; NASSU, B. T.; MINETTO, R. A video-based system for vehicle speed measurement in urban roadways. **IEEE Transactions on Intelligent Transportation Systems**, IEEE, p. 1–12, 2016. Acessado em: 2016.12.07. Disponível em: <http://ieeexplore.ieee.org/document/7576700/>.

MINETTO, R. et al. Snoopertext: A multiresolution system for text detection in complex visual scenes. In: **Proceedings of the IEEE International Conference on Image Processing (ICIP)**. Hong Kong: [s.n.], 2010. p. 3861–3864.

NABOUT, A. A.; TIBKEN, B. Object shape description using haar-wavelet functions. In: IEEE. **Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on**. [S.l.], 2008. p. 1–6.

PERRONNIN, F. et al. Large-scale image retrieval with compressed fisher vectors. In: IEEE. **Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on**. [S.l.], 2010. p. 3384–3391.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. Bioinfo Publications, 2011.

QADRI, M. T.; ASIF, M. Automatic number plate recognition system for vehicle identification using optical character recognition. In: IEEE. **2009 International Conference on Education Technology and Computer**. [S.l.], 2009. p. 335–338.

RIAZ, F. et al. Invariant gabor texture descriptors for classification of gastroenterology images. **IEEE Transactions on Biomedical Engineering**, IEEE, v. 59, n. 10, p. 2893–2904, 2012.

ROBERT, C. Machine learning, a probabilistic perspective. **CHANCE**, Taylor & Francis, v. 27, n. 2, p. 62–63, 2014.

RUBLEE, E. et al. Orb: An efficient alternative to sift or surf. In: IEEE. **2011 International conference on computer vision**. [S.l.], 2011. p. 2564–2571.

RUSSAKOVSKY, O. et al. ImageNet Large Scale Visual Recognition Challenge. **International Journal of Computer Vision (IJCV)**, v. 115, n. 3, p. 211–252, 2015.

SANDE, K. E. Van de et al. Segmentation as selective search for object recognition. In: IEEE. **2011 International Conference on Computer Vision**. [S.l.], 2011. p. 1879–1886.

SERMANET, P. et al. Overfeat: Integrated recognition, localization and detection using convolutional networks. **arXiv preprint arXiv:1312.6229**, 2013.

SZEGEDY, C. et al. Going deeper with convolutions. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2015. p. 1–9.

SZEGEDY, C. et al. Rethinking the inception architecture for computer vision. **arXiv preprint arXiv:1512.00567**, 2015.

WANG, L.; HE, D.-C. Texture classification using texture spectrum. **Pattern Recognition**, Elsevier, v. 23, n. 8, p. 905–910, 1990.