Very Deep Convolutional Networks for Large-Scale Image Recognition

Recognizing Traffic Cone

Author: Evandro Dessani
Pelletizing Department of VALE
UFES – Computer Science – LCAD
Vitoria ES, Brazil
evandro.dessani@vale.com

Author: Eduardo Frigini de Jesus
Pelletizing Department of VALE
UFES – Computer Science – LCAD
Vitoria ES, Brazil
eduado.frigini@vale.com

***Abstract* — In this work, we created a CNN (Convolutional Neural Network) for identification of traffic cones with TensorFlow and Keras. We verify what happens to the depth of the convolution network in its precision in the large-scale image recognition configuration. We had better explain the hardware and software requirements needed to build the network. Finally, we show how to set up a VGG 16 network.**

Keywords – Convolutional Neural Network; recognize; image; deep learnig

I. INTRODUCTION

Convolutional networks have greatly improved their performance due to hardware evolution, with powerful GPUs, graphics cards, initially made for gaming and computer graphics and now used on a large scale for deep neural network training. Moreover, this success is due to the huge databases cataloged with hundreds and even thousands of images that enable the training of such neural networks, such as ImageNet (Deng et al., 2009). In addition, competitions with different network architectures have proved very useful for improving performance and comparing different architectures. A very famous competition is the ImageNet Wide-ScaleVisual Recognition Challenge (ILSVRC) (Russakovsky et al., 2014).

Nowadays, autonomous cars are becoming more and more popular and will soon be the reality in our daily lives, with greater accuracy and even reducing traffic accidents.

With this perspective in mind, VALE S.A. and UFES (Federal University of Espirito Santo) in partnership are developing a joint work of an autonomous truck that can humidify the paved roads of the TUBARÃO complex.

In view of the project, we assume that the identification by neural networks of traffic cones, initially, as well as safety fencing, PARE, and SIGA signs will be of great importance for the autonomy of this truck.

We use Very Deep Convolutional Networks for Large-Scale Image Recognition to distinguish transit cones.

II. THE HARDWARE REQUERIMENTS

A. CPU vs GPU

With the entire spotlight on free frameworks for Big Data processing such as Hadoop and Spark, most professionals are not familiar with the concept of using GPUs (Graphical Processing Units) to process Big Data analytic solutions. When talking about GPUs, the vast majority of people who work with technology associate the term with the context of games or supercomputers. Many believe that the choice of GPUs as a hardware option to process Analytics applications is very exotic, but while there are challenges, the fact is that more and more GPUs have been used for advanced analysis using Machine Learning algorithms, especially Deep Learning. Let's start by understanding the difference between CPU and GPU.

As were originally sieves to render GPU graphics, but because of their high performance and low cost, they have become the new standard for image processing. Its facilities: restoration of images, segmentation, filtering, interpolation and reconstruction. The GPU computer chips are printed in the form of mathematical calculations in a fast and parallel way.

To see a difference between CPU and GPU, see the image below. The CPU consists of "cores" optimized for sequential serial processing, while a GPU has a parallel architecture of thousands of smaller, more efficient "cores" to handle multiple tasks simultaneously, as in Figure 1.

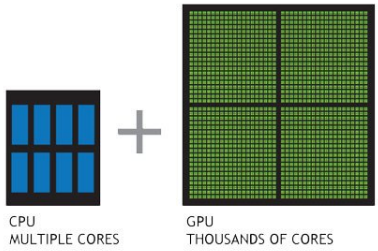

Figure 1 - CPU's have few "cores" with cache. GPU's have thousands of "cores" that can handle thousands of threads simultaneously.

B. How the GPU's works

GPU accelerated computing is the use of GPUs in conjunction with CPUs in order to accelerate scientific analysis, analytics, engineering, and applications. What the GPU does is receive from the application the processing that requires more intensive use of computing, as in Figura 2. The GPU receives a portion of the application, while the CPU receives the remainder. Parallel to execution, applications get faster. And with the adoption of GPU processing by Apple (with OpenCL) and by Microsoft (with DirectCompute), it's a matter of time for GPU processing to become standard in the market.

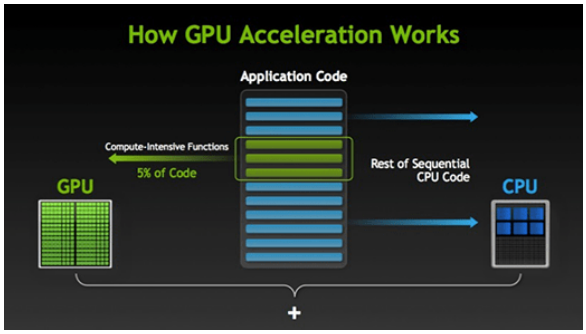

Figure 2 - The GPU receives a portion of the application, while the CPU receives the remainder. Parallel to execution.

III. THE SOFTWARE REQUERIMENTS

We could use a large list of softwares, like: Caffe, Caffe2, Chainer, Microsoft Cognitive Toolkit, Matlab, Mxnet, PaddlePaddle, Pytorch, TensorFlow, Theano, Keras, Torch, WolFram, among other. We decide to use TensorFlow with Theano and Keras.

For this, we list the sequence to install these software's:

1-Install Anaconda:

Access the site http://anaconda.com/download download the respective version to your OS and the version of Python

2-Linux Ubuntu 16:

Python 3.6 https://repo.continuum.io/archive/Anaconda3-5.0.1-Linux-x86_64.sh Python 2.7 https://repo.continuum.io/archive/Anaconda2-5.0.1-Linux-x86_64.sh Open a terminal, access the location where the file was downloaded and follow with the following command bash ~ / Downloads / Anaconda3-5.0.1-Linux-x86_64.sh (version 3.6 of Python)

3-Python for Windows

Python 3.6 https://repo.continuum.io/archive/Anaconda3-5.0.1-Windows-x86_64.exe Python 2.7

https://repo.continuum.io/archive/Anaconda2-5.0.1-Windows-x86_64.exe To install, just double-click on the installer and follow the

From this moment, we will cover the steps only for linux as an operating system.

4-Install-CUDA-Cudnn-support-GPU

If you want to run the training with GPU features you need to install the CUDA and Cudnn, packages the versions of these are very important so that everything works correctly as of the moment the last available version of the tensorflow is 1.4.

We used the Cuda and Cudnn versions that the page recommends as supported versions remembering that the video card must be compatible with cuda and have the nvidia drivers installed

5-CUDA 8 GA2

To facilitate the installation we will use wget to open a terminal to access wherever the file is downloaded / home / user / Downloads mkdir Cuda cd Cuda wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/ local_installers / cuda_8.0.61_375.26_linux-run as soon as the download is completed give the execution right to the sudo file chmod + x filename sudo sudo ./nameArchiveBased the same situation for the patch of this version wget https://developer.nvidia.com/ / cuda / 8.0 / Prod2 / patches / 2 / cuda_8.0.61.2_linux-run as soon as the download is finished give the executable right to the sudo file chmod + x filenamePatchDownloaded sudo ./namePatchFileName Then we must download the cuda files and extract -los wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v6/prod/8.0_20170307/Ubuntu16_04_x64/libcudnn6_6.0.20-1+cuda8.0_amd64-deb sudo dpkg -i filenameDownloaded

6-Path

The next step is to include the path of these packages in the PATH sudo nano ~ / .bashrc insert at the end of the file the following lines export PATH = / usr / local / cuda-8.0 / binPATH: +: $ PATHexportLDLIBRARYPATH = / usr / local / cuda-8.0 / lib64 PATH: +: $ PATHexportLDLIBRARYPATH = / usr / local / cuda-8.0 / lib64 {LD_LIBRARY_PATH: +: $ {LD_LIBRARY_PATH}}

sudo apt-get install libcupti-dev

7-Installing Packages

Create a specific environment for this terminal within the terminal conda create -n ufes_lcad python = 3.6 source activate ufes_lcad install the basic packages that we will work with install keras conda install matplotlib conda install numpy conda install pandas conda install scikit-learn pip install --ignore -installed --upgrade https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow_gpu-1.4.0-cp36-cp36m-linux_x86_64.whl

8-Starting Jupyter

Inside the terminal activate ufes_lcad anaconda-navigator or just jupyter click on new -> python 3

IV. Convolutive Neural Networks

LeCun et al. [1], in which the authors developed a neural architecture known as LeNet5, used to recognize handwritten digits, first proposed convolutional Neural Networks (CNNs) in 1998. At the time, such architecture established a new state of art by achieving 99.2% accuracy in the MNIST database [2]. Years later, Krizhevsky et. al [3] established a milestone in the area by proposing AlexNet, the winning architecture of the ImageNet challenge [4]. Since then, most popular application related to CNNs have been recognizing patterns in images, however, several papers have been applying CNNs in other types of tasks [5].

CNNs are similar to traditional neural networks: both are composed of neurons that have weights and bias that need to be trained. Each neuron receives some inputs, applies the scalar product of inputs and weights in addition to a non-linear function. In addition, both have the last layer all connected and all the artifacts used to improve the traditional neural network are also applied in this layer. So what is the advantage of using a CNN? A CNN assumes that all inputs are images, which allows you to encode some properties in the architecture. Traditional neural networks are not scalable for images, since they produce a very high number of weights to be trained [6].

A CNN is composed of a sequence of layers. In addition to the input layer, which normally consists of an image with width, height and depth (RGB), there are three main layers:

- ✓ convolutional layer,
- ✓ pooling layer and
- ✓ fully connected layer.

In addition, an activation layer (usually a ReLu function) is common after a convolutional layer. These layers, when sequenced (or stacked), form an architecture of a CNN, as shown in Figure 3.

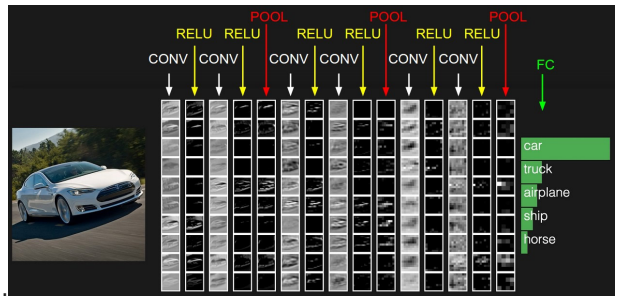

Figure 3 - example of an architecture of a CNN

1) The convolutional layer

The convolutional layer is the most important layer of the network. It carries out the heaviest part of computational processing. This layer is composed of a set of filters (or kernels) capable of learning according to a training. Filters are small matrices (for example, 5 x 5 x 3) composed of real values that can be interpreted as weights. These filters are convolved with the input data to obtain a feature map. These maps indicate regions in which specific characteristics in relation to the filter are found at the entrance. The actual values of the filters change throughout the training (as well as the weights of a traditional neural network), causing the network to learn to identify significant regions to extract characteristics from the data set.

The convolution between a filter and the image is illustrated in Figure 4. In this example, a 4 x 4 x 3 image and a 2 x 2 x 3 filter are shown. The convolution is performed through the scalar product of a region of the image size of filter by the filter itself. Then the filter is slid to another region and the scalar product is performed again until the entire image is traversed. Notice that a different dimension of the filter, which in this example is represented by the same channel color (RGB), convolves each channel. In this example, the filter slides on the image by moving 1 pixel to the side or down. This sliding is controlled by the parameter known as 'stride', which in this case is 1. This value can be changed, however it must respect the image limits. If the desired filter and stride size is not possible for the image, you can use 'zero-padding', which is nothing more than adding zeros to the edge of the image to make filter sliding possible. Both 'stride' and 'zero-padding' are project parameters that must be analyzed by the designer. Finally, it is worth noting that more than one filter per convolution layer may exist (and usually exist). In this way, each filter results in a three-dimensional output, as in Figure 4.

In the convolution results matrices the activation function is applied. The most commonly used function is the ReLU (linear rectification unit), which is simply to apply the function max (0, x) on each element of the convolution result. In the example of Figure 4, all elements are greater than zero, so the result of ReLU is the values themselves. Finally, since the weights used by each filter are the same in all regions, they are known as shared weights. This feature considerably reduces.

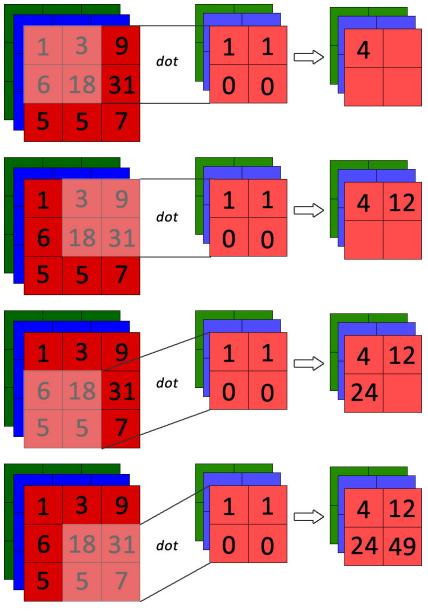

Figure 4 - Example of convolution between the image and a filter

2) The pooling layer

It is very common after a convolution layer there is a layer of pooling. The pooling technique is used to reduce the spatial size of the matrices resulting from the convolution. Consequently, this technique reduces the amount of parameters to be learned in the network, contributing to the control of overfitting. The pooling layers operate independently on each of the channels of the convolution result. In addition, you must first determine the size of the filter and stride to perform the pooling. Figure 5 shows the realization of pooling considering that the convolution result is a 4 x 4 x 3 matrix, the pooling filter covers 2 x 2 regions on each channel, with stride equal to 2. In addition, the aggregation function is equal to max(X), that is, it chooses the highest value of the region.

As can be seen, the pooling layer does not reduce the number of channels but the amount of elements in each channel. In addition to maxpooling, other functions, such as average values, can be used. However, the max function has been getting better results, since neighboring pixels are highly correlated.

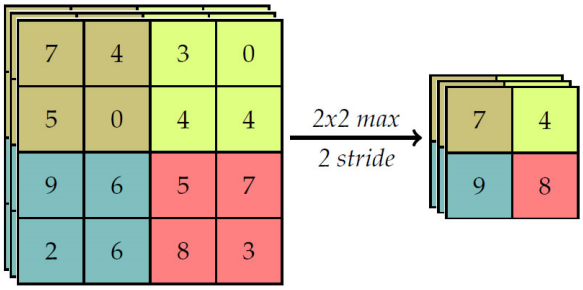

Figure 5 - example of max pooling the a 4 x 4 image

3) The fully connected layer

Unlike the convolutional layer, in which weights are connected only in one region, the fully connected layer, as the name itself suggests, is completely connected to the previous layer. They are typically used as the last layer of CNN and work in the same way as traditional neural networks. Therefore, the same techniques to improve the performance of an RNA, such as Dropout for example, are also applicable in this layer.

Since the fully connected layer comes after a convolutional or pooling layer, it is necessary to connect each element of the convolution output matrices to an input neuron. Figure 6 shows the connection of a convolutional layer with a fully connected layer. It is possible to observe that the 48 maps of characteristics 4 x 4 are placed of linear form forming 768 entrances for layer totally connected, that in turn, has 500 hidden neurons that result in 10 exits for network. At this point, it is possible to visually observe a traditional neural network.

In the output units, which in the example of Figure 6 are 10, a softmax function is used to obtain the probability of a given entry belonging to a class. At this point the backpropagation supervised training algorithm is performed, as well as in a traditional neural network. The error obtained in this layer is propagated so that the weights of the filters of the convolutional layers are adjusted [1]. In this way, shared weight values are learned throughout the training.

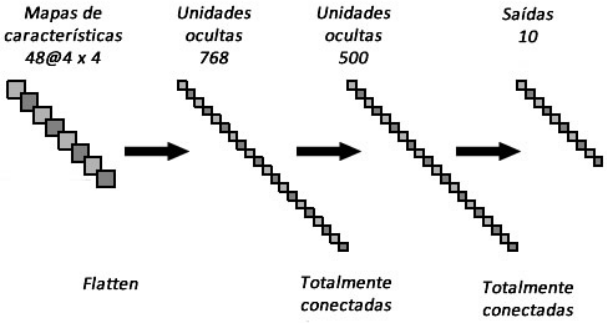

Figure 6 - illustration of a fully connected layer

V. THE ARCHITECTURE OF A CNN

The architecture of a CNN is related to the way the layers described above are organized. In fact, the designer can organize them in whatever way he deems appropriate. Most architectures use an input layer, of course, followed by a block of N convolutional layers with ReLU activation connected to a pooling layer. This block is repeated M times along the network, which in the end is connected to a fully connected layer to determine the final classification. In addition, in the architecture assembly parameters such as filter size, stride and zero padding are also defined. All of these choices directly impact the amount of weights that the network must train and consequently how much computing is required for these values to be well adjusted.

There are several well-known architectures in the area of convolutional networks. Three of the most used are described in the sequence:

- **LeNet-5:** was developed by LeCun et al. [1] and was the first successful CNN application. It was used to classify digits, because of this, much used in tasks such as reading postal codes and the like. Figure 7 illustrates the architecture of a LeNet-5, which has an input layer with 32 x 32 pixels images and a further 7 layers of trainable weights. The convolutional layers use 5 x 5 x 3 and 6 x 6 x 3 filters to compute feature maps and 2 x 2 x 3 filters for pooling. Finally, in addition to the output layer that has connection for 10 classes, the network has two fully connected layers of 120 and 84 neurons.

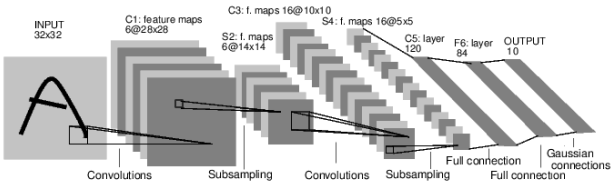

Figure 7 - the architecture of a Lenet-5

- **AlexNet**: AlexNet architecture was the work that popularized CNNs for computer vision. Proposed by Krizhevsky et al. [3], was the winning methodology of the ImageNet challenge in 2012. Its architecture is quite similar to LeNet-5, but it is deeper, with more convolutional layers, and has many more feature maps, as shown in Figure 8. AlexNet receives as input 224 x 224 pixels per channel. In the first convolution layer, it uses a filter 11 x 11 x 3, in the second 5 x 5 x 3 and in the third a 3 x 3 x 3. In addition, the third, fourth and fifth layers are connected without the use of pooling. Finally, the network has two layers fully connected with 2048 neurons each and an output layer with 1000 neurons, number of classes in the problem. It is worth mentioning that AlexNet was the first network to use Dropout [7] to assist in the training of the fully connected layer. In addition, training of the entire network was performed using two GPUs. As can be seen in Figure 8, the network is divided into two parts; one GPU executed the top and the other the bottom part of the network.

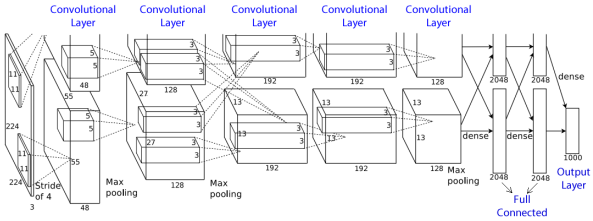

Figure 8 - the architecture of an AlexNet

- **VGGNet:** the VGGNet architecture was proposed by Simonyan and Zisserman [8] and had as main contribution to show that the depth of the network is a critical component for a good performance. The standard VGGNet model has several convolution layers applying 3 x 3 x 3 filters and maxpooling with 2 x 2 x 3 filters. Figure 9 shows a comparison between an AlexNet and a VGGNet. It is possible to observe a layer of pooling always after two convolution. In addition, the network has three layers fully connected beyond the output layer. Due to its depth, VGGNet is very expensive computationally and requires a lot of memory to compute over its parameters (~ 140M).

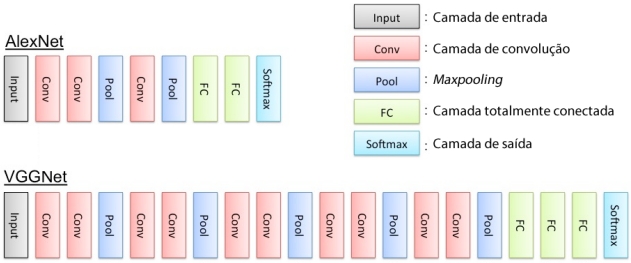

Figure 9 - comparison between the AlexNet and VGGNet architectures

A. Considerations and frameworks

With the main CNN blocks, it is possible to build the architecture that best meets the needs. However, it is not always easy to find a pattern and parameters that perfectly fit the problem. The great advantage of CNN is that it is able to extract the characteristics of the input images without the need for a pre-processing in them. The more convolution layers, the more characteristics are drawn. However, just like in traditional neural networks, one must be careful with overfiting and with useless features. For example, Lenet is applied for character identification problems, so it is small and performs its function almost optimally. VGGnet is already applied to images with more than 1000 labels, so there are many more features to be extracted than to the characters at hand. If we apply to VGGNet for MNIST we are "killing a cockroach with a bazooka" because there is no need for such a large network for such a problem. Unfortunately, there is no secret formula for choosing all these parameters (stride, layers, zero padding, etc.). This ability only comes with the practice and use of CNN.

It is possible to use CNNs very simply and quickly, through the use of frameworks. Some use faster, just choose the parameters, enter the training and run data, and others that need a bit more elaboration. In addition, it is possible to use GPUs according to the tool used and availability on the machine. In this article we suggest three frameworks:

• Keras: simpler and more direct use. Because of this, it is less malleable to modifications. In addition, it is necessary to have the tensorflow installed in the machine. The language used is Python.

• Tensorflow: speaking of it, this is our second recommendation. Framework developed by Google that massively used in the field of machine learning. In it CNN's main build blocks are implemented and you can use them to build your architecture. It will take a little more practice to master them, but it is more powerful than Keras (actually Keras is built on top of Tensorflow, that is, it's like a facilitator for it). It also uses Python (although it is expanding to new languages).

• Caffe: this is for lovers of C / C ++, language used by the library. In it is possible to implement the architectures much like Keras.

Our choice was for a VGG16, due to the ability to identify different objects. We limited CNN to 16 layers because of the limited time and computational capacity we had. We used TensorFlow and Keras to train and test the network. These choices are detailed below.

VI. Mounting VGG Network 16

A. Architecture

Constructing the first layer. We define that this layer will produce 64 output filters, its kernel is 3 x 3 the stride is 1 by 1 the input will be 224 by 224 color, therefore 3 channels and the activation function of this layer will be ReLu. Following sequentially configure the second layer, which takes as input the output of the previous layer and so we do not need to declare this value, here we still have 4 output filters, 3 x 3 kernel and stride is 1 by 1. We continue to the first layer of max Pooling, this layer has a 2 by 2 size kernel pool and a 2 by 2 stride. With this we conclude the first block of VGG 16, I will consider the end of each block always after a pooling layer. In the second block of VGG 16 there is an increase in the number of outputs of the layers in 2x the other parameters are the same, this increase occurs whenever there is a pooling and has multiplication factor 2 limited to 512. We proceed to the second layer of max Pooling, this layer has a 2 by 2 size kernel pool and a 2 by 2 stride also. In the third block of VGG 16 there is an increase in the number of outputs of the layers in 2x and there is the addition of another convolutional layer before the pooling we follow for the third layer of max Pooling, this layer has a 2 by 2 size kernel pool and a 2 by 2 stride. In the fourth block of VGG 16 there is an increase in the number of outputs of the layers by 2x the other parameters are the same we proceed to the third layer of max Pooling, this layer has a 2 by 2 size kernel pool and a 2 by 2 stride. In the fifth block of VGG 16 all parameters are the same as the fourth layer, the filter values are not multiplied by 2 because it has reached we have the 512 ceiling we continue to the fourth layer of max pooling. This layer has a 2 by 2 size kernel pool and a 2 by 2 stride at the moment we prepare the output for the format that the first FullyConnected layer input needs to be 3 layers FC. The first has an input size of 4096 and Relu activation function, shortly after the first layer FC is done a dropout with 50% rate. The second FC layer has the same configuration as the first. Shortly after the second layer FC is made a further dropout with 50% rate The third layer in the original network VGG16 has input of size 1000 because it classifies 1000 classes because of our need to classify only 3 classes my entry number to the third layer will be the activation function of the third layer to softmax. As in the Figure 10.

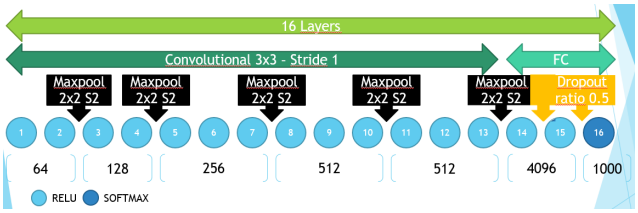

Figure 10 – The VGG16 architecture

B. Training

Before starting the training of the network we need to define where the photos that will be used for training and where are the photos that will be used to test to increase the number of samples of photos we use the feature of keras that increases / decreases the photo, turns and etc.

Here we also define the format that will be the input of the first network layer, the input pattern for VGG16 224 x 224 x 3, the default batch of VGG16 is 256.

The parameters are:

- ✓ Image input sizes = 224 x 224
- ✓ Batch Size = 256
- ✓ Momentum 0.9
- ✓ Dropout Ratio = 0.5
- ✓ Penalty Multiplier = 0.0005
- ✓ Initial Learning Rate = 0.01
- ✓ Final Learning Rate = 0.00001
  - o The learning rate was set to decrease by a fator of 10 when the validation set accuracy stopped improving
  - o In total, the learning rate was decreased 3 times, and the learning was stopped after 370K iterations (74 epochs).
- ✓ Initialization Weights
  - o First four Convolutional Layers and the last three conected layers was initialized with the layers of netconf A
  - o The intermediate layers were initialized randomly

C. Testing

We then define where the directory is containing the separated dataset in folders with the following hierarchy inside. The dataset folder we have two more folders with name training and test inside each of the training and test folders. We have the amount of folders equivalent to the classes in my case 3, ie I have 3 folders inside the training folder and 3 more folders inside the test folder they must have the same name and contain the class that will be trained.

D. Implementations Details

The code in TensorFlow is describe below:

```
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense
from keras.layers import Dropout
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
width, height, batch = 32, 32, 8
sgd = SGD(lr=0.01, decay=5e-4, momentum=0.9, nesterov=True)
```

```
train_datagen = ImageDataGenerator(rescale = 1./255, shear_range = 0.2, zoom_range = 0.2, horizontal_flip = True)
test_datagen = ImageDataGenerator(rescale = 1./255)
training_set = train_datagen.flow_from_directory('dataset/training_set', target_size = (width, height), batch_size = batch, class_mode = 'categorical')
test_set = test_datagen.flow_from_directory('dataset/test_set', target_size = (width, height), batch_size = batch, class_mode = 'categorical')
classifier = Sequential()
classifier.add(Conv2D(filters = 64, kernel_size = (3, 3), padding='same', strides=(1,1), input_shape = (width, height, 3), activation = 'relu'))
classifier.add(Conv2D(filters = 64, kernel_size = (3, 3), padding='same', strides=(1,1), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2), strides=(2,2)))
classifier.add(Conv2D(filters = 128, kernel_size = (3, 3), padding='same', strides=(1,1), activation = 'relu'))
classifier.add(Conv2D(filters = 128, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2), strides=(2,2)))
classifier.add(Conv2D(filters = 256, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 256, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 256, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2), strides=(2,2)))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2), strides=(2,2)))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(Conv2D(filters = 512, kernel_size = (3, 3), padding='same', strides=(1,1), activation= 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2), strides=(2,2)))
classifier.add(Flatten())
classifier.add(Dense(units = 4096, activation = 'relu'))
classifier.add(Dropout(0.5))
classifier.add(Dense(units = 4096, activation = 'relu'))
classifier.add(Dropout(0.5))
classifier.add(Dense(units =3, activation = 'softmax'))
classifier.compile(optimizer = 'sgd', loss = 'categorical_crossentropy', metrics = ['accuracy'])
classifier.fit_generator(training_set, steps_per_epoch = 1000, epochs = 2, validation_data = test_set, validation_steps = 250, use_multiprocessing=False)
```

E. Classification Experiments

This experiment was well successes and we identify different kind of traffic cones in the dataset used.

Since the number of categories is rather large, it is conventional to report two error rates: top-1 and top-5, where the top-5 error rate is the fraction of test images for which the correct label is not among the five labels considered most probable by the model. The table 1 shows some predictions made by our model on a few test images.

Table 1: **Multiple ConvNet fusion results.**

Combined ConvNet models	Error		
	top-1 val	top-5 val	top-5 test
ILSVRC submission			
(D/256/224,256,288), (D/384/352,384,416), (D/[256;512]/256,384,512) (C/256/224,256,288), (C/384/352,384,416) (E/256/224,256,288), (E/384/352,384,416)	24.7	7.5	7.3
post-submission			
(D/[256;512]/256,384,512), (E/[256;512]/256,384,512), dense eval.	24.0	7.1	7.0
(D/[256;512]/256,384,512), (E/[256;512]/256,384,512), multi-crop	23.9	7.2	-
(D/[256;512]/256,384,512), (E/[256;512]/256,384,512), multi-crop & dense eval.	**23.7**	**6.8**	**6.8**

VII. Comparation of State of Art

Comparison with the state of the art in ILSVRC classification. Our method is denoted as "VGG". Only the results obtained without outside date are reported. As Table2.

Table 2: **Comparison with the state of the art in ILSVRC classification.**

Method	top-1 val. error (%)	top-5 val. error (%)	top-5 test error (%)
VGG (2 nets, multi-crop & dense eval.)	**23.7**	**6.8**	**6.8**
VGG (1 net, multi-crop & dense eval.)	24.4	7.1	7.0
VGG (ILSVRC submission, 7 nets, dense eval.)	24.7	7.5	7.3
GoogLeNet (Szegedy et al., 2014) (1 net)	-	7.9	
GoogLeNet (Szegedy et al., 2014) (7 nets)	-	**6.7**	
MSRA (He et al., 2014) (11 nets)	-	-	8.1
MSRA (He et al., 2014) (1 net)	27.9	9.1	9.1
Clarifai (Russakovsky et al., 2014) (multiple nets)	-	-	11.7
Clarifai (Russakovsky et al., 2014) (1 net)	-	-	12.5
Zeiler & Fergus (Zeiler & Fergus, 2013) (6 nets)	36.0	14.7	14.8
Zeiler & Fergus (Zeiler & Fergus, 2013) (1 net)	37.5	16.0	16.1
OverFeat (Sermanet et al., 2014) (7 nets)	34.0	13.2	13.6
OverFeat (Sermanet et al., 2014) (1 net)	35.7	14.2	-
Krizhevsky et al. (Krizhevsky et al., 2012) (5 nets)	38.1	16.4	16.4
Krizhevsky et al. (Krizhevsky et al., 2012) (1 net)	40.7	18.2	-

VIII. Conclusions

In this work, we evaluated very deep convective networks (up to 16 layers) for the classification of images of traffic cones. We introduce the software and hardware requirements needed to run a CNN. We explain step by step how a convolutional network works: convolutional layer, pooling layer and fully connected layer. We talked about the frameworks used for network programming. We show the proposed architecture that was a VGG 16 network and detail its implementation.

It has been shown that the depth of the representation is beneficial to the accuracy of the classification. The results were satisfactory showing that our network is capable of being trained and it is possible to identify transit objects with good precision.

Acknowledgment

This work was supported by VALE in partnership with UFES of the laboratory LCAD in development with the work of autonomous vehicles.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998

[2] L. Yann, C. Corinna, and B. Christopher, "The MNIST database of handwritten digits," disponível em: https://www.kaggle.com/c/thenature-conservancy-fisheries-monitoring/details/evaluation, visitado em 8 de dezembro de 2016.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Advances in neural information processing systems, 2012, pp. 1097–1105.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, 2009, pp. 248–255.

[5] E. Gibney, “Google ai algorithm masters ancient game of go,” Nature, vol. 529, pp. 445–446, 2016.

[6] F.-F. Li, A. Karpathy, and J. Johnson, “Cs231n: Convolutional neural networks for visual recognition,” 2015

[7] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting.” Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, 2014.

[8] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” arXiv preprint arXiv:1409.1556, 2014.