DARTS: Deceiving Autonomous Cars with Toxic Signs

Antonino Di Maggio (1702628) and Leonardo Salvucci (1601997)

Sapienza - University of Rome

April 14, 2020

Abstract

Recent studies show that the cutting edge deep neural networks are vulnerable to contradictory examples, deriving from small magnitude perturbations added to the input. With the advent of self-driving machines, the contradictory example, as we can imagine, can generate many complications: a car can interpret a signal incorrectly and generate an accident. In the following report, we want to analyze and test the problem, showing that it is possible to generate specific perturbations to the input images to confuse the model and, in some way, force the network prediction.

I. Introduction

In the following report we will explain our project in detail, aimed at explaining and implementing a good part of an existing project described on the proposed paper DARTS: Deceiving Autonomous Cars with Toxic Signs, written by Chawin Sitawarin, Arjun Nitin Bhagoji, Arsalan Mosenia, Prateek Mittal, from the Princeton University, and Mung Chiang, from Purdue University.

The paper explained how to modify an existing traffic signal (or another generic signal), to deceive a model of an autonomous driving car. In the paper various kind of attacks are presented:

- **White box attack:** in this kind of attack, we have a trained model and its weights, so the aim is to create an adversarial image in order to mislead the specific model.

- **Black box attack:** in this case, we do not have a model, so the aim is to create an adversarial image that has to seems like another one for a generic model.

The attacks mentioned can still be distinguished in:

- **Targeted attack:** in this case, the aim is to choose a specific class of traffic signal such that the images (other signals, logos or custom signs) we are going to modify have to be recognized by the model as that.

- **Untargeted attack:** in this case the input image has to be a picture of a traffic sign and the aim is just to generate a wrong prediction, without specify a specific output class.

Or even:

- **In-distribution attack:** the in-distribution attack consist in modify an image representing a traffic signal X, in order to seems like another traffic signal. It is called in-distribution because we have to modify an image that the model is trained to recognize in another image that the model knows.

- **Out-of-distribution attack:** in this case the image is an image for which the model is not trained, in our case we have to cases of attacks:

  - **Logo attack:** the image is a commercial logo (Burger King logo, KFC logo, etc.).
  - **Custom signs attack:** in this case we have an image representing a signal with no pattern, but just with plain color.

We choose to implement the white box attack, for both the targeted and the untargeted attack and, for the targeted attack, both the in-distribution and out-of-distribution attacks. For the untargeted attack, instead, we implemented only the in-distribution attack, because it would have no sense to implement the out-of-distribution attack.

We implemented two functions to perform two different kinds of attacks:

- **Gradient attack**;
- **Iterative attack**.

That we are going to present later in this report.

II. The Dataset

We use the GTSRB - German Traffic Sign Recognition Benchmark dataset. It is a multi-class, single-image classification challenge held at the International Joint Conference on Neural Networks (IJCNN) 2011.

i. Structure

The GTSRB has the following properties:

- Single-image, multi-class classification problem;
- 43 classes;
- More than 50,000 images in total;
- Large, lifelike database;
- Reliable ground-truth data due to semi-automatic annotation;
- Physical traffic sign instances are unique within the dataset.

In the train sub-directory, there are 43 directories (numbered from 0 to 42), each of this directory represent a different kind of traffic signal, in this way:

ID	Traffic Sign Description
0	Speed limit (20km/h)
1	Speed limit (30km/h)
2	Speed limit (50km/h)
3	Speed limit (60km/h)
4	Speed limit (70km/h)
5	Speed limit (80km/h)
6	End of speed limit (80km/h)
7	Speed limit (100km/h)
8	Speed limit (120km/h)
9	No passing
10	No passing for vehicles over 3.5 metric tons
11	Right-of-way at the next intersection
12	Priority road
13	Yield
14	Stop
15	No vehicles
16	Vehicles over 3.5 metric tons prohibited
17	No entry
18	General caution
19	Dangerous curve to the left
20	Dangerous curve to the right
21	Double curve
22	Bumpy road
23	Slippery road
24	Road narrows on the right
25	Road work
26	Traffic signals
27	Pedestrians
28	Children crossing
29	Bicycles crossing
30	Beware of ice/snow
31	Wild animals crossing
32	End of all speed and passing limits
33	Turn right ahead
34	Turn left ahead
35	Ahead only
36	Go straight or right
37	Go straight or left
38	Keep right
39	Keep left
40	Roundabout mandatory
41	End of no passing
42	End of no passing by vehicles over 3.5 metric tons

With the dataset, we even downloaded the images contained in the directory *Original_Traffic_Signals_samples*, which contains some samples in high definition, these are the images that we are going to modificate in order to be detected as other signals.
We even have to other directories:

- Logo_samples;
- Custom_Sign_samples.

ii. Data augmentation

The dataset, originally, was highly unbalanced, as you can see in the following histogram:

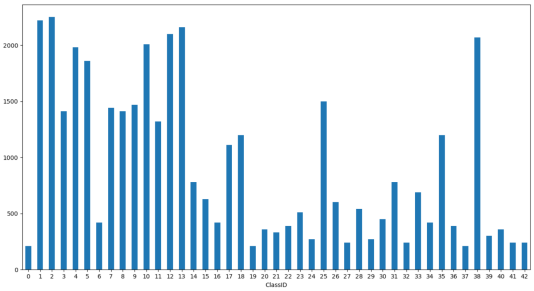

Figure 1: *Dataset histogram*

So, in order to obtain a better training, we had to balance it. For this aim, we created the file called *data_augmentation.py*, containing four functions to apply a transformation to the existing images:

- ***apply_flip:*** in this function, we created two arrays to identify the signals that can be flipped, horizontally or vertically, without changing theirs meaning. Than, if a signal is flippable, we apply a flip of the image of the signal;

(a) *Original* **(b)** *Augmented*

- ***apply_brightness:*** apply a random changing of the brightness of the signal;

(a) *Original* **(b)** *Augmented*

- ***apply_rotate_image:*** apply a random rotation of the signal;

(a) *Original* **(b)** *Augmented*

- ***apply_projection_transform:*** apply a random perspective transformation of the signal;

(a) *Original* **(b)** *Augmented*

and these other to manage them:

- ***add_data:*** if a class contains less than target size (target_size = 2000) images, then one of the transformations is randomly applied to the original image and saved in the dataset;
- ***rem_data:*** otherwise, if a class contains more than target size images, we remove some of the existing images to balance the dataset classes;

Of course, the functions mentioned are only applied to the original images, to avoid increasing the images too many times and no longer being able to manage the changes. Then, the images are read, kept locally, and randomly modified until they reach the expected cardinality.

At this point, we obtained a balanced dataset, increasing it, from 51.839 to 80.000 images (2000 for each class).

III. The model

i. Model summary

We implemented the withe box attack, so we needed a pre-structured and trained model to deceive, this is the summary of the model that we built:

- **Input layer:** We built an input layer to avoid to have to preprocess images that are too large, risking to saturate the memory of our computer, and even to have uniform images to give as input for our model. In order to do this, we passed to the model an *INPUT_SHAPE* = (32, 32, 3).

- **Convolutionary layers:** We used 3 convolutional layers, using 32 filters with 5x5 receptive field, to specify the strides of the convolution along the height and the width. This layers create a convolution kernel that is convolved with the layer input to produce a tensor of outputs. Each convolutional layer is followed by a *ReLU* activation function.

- **Dropout layers:** This layers randomly sets a fraction rate of input units to 0 at each update during the training time, this is very useful to prevent overfitting.

- **Pooling layers:** this layers help the model to remove un-useful informations, this is needed because, very often, neighboring elements contains very similar informations.

These three layers are repeated three times and then we insert a flatten layers for each pool of the previous layers, its aim is to change the shape from the 2D matrix produced by the previous layers into the correct shape to be interpreter by a dense layer. Than, before the dense layer, we merge the outputs of the three flatten layers. At the end, just before the output, we have a dense layer, followed by another dropout layer and a last dense layer, that correspond to the output layer of our model.
Optimizer: as optimizer, we used the *Adam*, with a learning rate equal to 0.0001.

ii. Training

We trained the model with the augmented and balanced dataset, setting the number of epochs equals to 100. We checked if for 6 (patience=6) epochs the train accuracy do not improve, in that case, the training would be stopped.
The model reach very fast a very high accuracy for both train and validation, and indeed, at the epoch n.48, we obtained this following results:

RESULTS		
	Train	Validation
Accuracy	0.9973	0.9941
Loss	0.0431	0.0632

iii. Weights

To avoid to re-train the model at every execution of the attacks, and even to charge every time the whole trained model, once we obtained the trained model, we saved its weights in a *.hdf5* file named *mltscl_cnn.hdf5*.

iv. Model's test

We tested the model on the test set proposed by the dataset. To execute it, as in the train, we used image data generator, to process the images and pass them to the model for prediction. We have achieved an excellent result reaching an accuracy of **96.41%**.

IV. Fast Gradient attack

i. Overview

The **fast gradient attack** works by using the gradients of the neural network to create an adversarial example. For an input image, the

method uses the gradients of the loss with respect to the input image to create a new image that maximises the loss. This new image is called the adversarial image.

ii. Untarget attack

This method computes an adversarial image by adding a pixel-wide perturbation of magnitude in the direction of the gradient. This perturbation is computed with a single step, thus is very efficient in terms of computation time:

$$x_{adv} = x + \varepsilon \cdot sign(\nabla x \quad J(\theta, x, y_{true})$$

where:
- x_adv : Adversarial image
- x: Original image
- ε: Small multiplier
- y: Original classification
- θ: Model parameters
- J: Classification loss function

iii. Target attack

In targeted attacks the attacker pretends to get the image classified as a specific target class, which is different from the true class. in this case in the direction of the negative gradient with respect to the target class:

$$x_{adv} = x - \varepsilon \cdot sign(\nabla x \quad J(\theta, x, y_{target})$$

where:
- x_adv : Adversarial image
- x: Original image
- ε: Small multiplier
- y: Target class
- θ: Model parameters
- J: Classification loss function

iv. Execution

To execute the attack we have implemented the following function:

$$fg(model, x, y, mask, target)$$

where:
- *model* : The trained model to attack
- x: A numpy array containing the samples to attack.
- *target*: True if it's a TARGET attack, False otherwise
- y: A numpy array containing the respective labels. It is important to note that if it's target attack the list contains always the same value (Target class), otherwise it contains, for each cell, the respective label.
- *masks*: The list of respective masks to restrict gradient update only on the image's portion that contain the signal, excluding the background (Optional).

The function returns a numpy array containing the adversarial images that we save in a specific folder.

v. Attack's description

First of all, we initialize the variable *x_adv*, that will contains our adversarial example, as a numpy array of zeros that as the same dimension of our input images plus the length of the magnitude list. Than we have to create a function that have to calculate the gradient of an input image, with respect of the Keras model. In order to do this we created a function called *gradient_fn*, that takes as input the model and return the gradient, using the gradient function provided by the *keras.backend* library and return the function the we need.

In order to perform the attack, we have to iterate through the input images in x and, for each image, we have to calculate its gradient using the function *grad_fn* that we created. Than we applicate the mask and normalize the gradient.
At the end, we create the perturbation according to a specific value and it returns the adversarial images clipped according the function *numpy.clip* with interval [0, 1], in this case, in the *x_adv* array, the values smaller that 0 are converted to 0, and the ones bigger than 1 are converted to 1.

V. Iterative attack

The **iterative attack** is more sophisticated in confront of the fast gradient one. It moves a benign sample in the gradient direction one small step at a time for n steps.

i. Execution

The main function to perform this attack is defined in this way:

$$iterative(model, x, y, mask, target)$$

where:
- *model* : The trained model to attack
- x: A numpy array containing the samples to attack.
- *target*: True if it's a TARGET attack, False otherwise
- y: A numpy array containing the respective labels. It is important to note that if it's target attack the list contains always the same value (Target class), otherwise it contains, for each cell, the respective label.
- *masks*: The list of respective masks to restrict gradient update only on the image's portion that contain the signal, excluding the background (Optional).

The function returns a numpy array containing the adversarial images that we save in a specific folder.

ii. Attack's description

The first part of this attack is very similar to the fast gradient attack, indeed: First of all, we initialize the variable *x_adv*, that will contains our adversarial example, as a *numpy* array of zeros that as the same dimension of our input images plus the length of the magnitude list.

Than we have to create a function that have to calculate the gradient of an input image, with respect of the Keras model. In order to do this we created a function called gradient_fn, that takes as input the model and return the gradient, using the gradient function provided by the *keras.backend* library and return the function the we need. In order to perform the attack, we have to iterate through the input images in x and apply the mask. But, at this point, we have to iterate again according to the number of steps we set (we set 40 steps).

iii. Observations

First attack have lower success rates when compared to the iterative methods in white box attacks, however when it comes to black box attacks the basic single-shot methods turn out to be more effective. The most likely explanation for this is that the iterative methods tend to overfit to a particular model.

VI. Attack settings

Before starting the attacks, we had to set some constants, in order to perform and store results correctly.

i. Variables' setting

It is important to find a good values for the magnitude such that the image can be modified nor too less (it would be very difficult to mislead the model), nor too much (it would be obvious, even for a human, that a sign is not the original one).

Then we create another variabile, called *attack_type*, we can assign to this variable three possibile values:

- ***IN_DISTRIBUTION:*** if we want to perform the in-distribution attack. In this case, if we are performing a targeted attack, we delete from the input samples the images having as label the target;
- ***LOGO:*** if we want to perform the logo attack;
- ***BLANK:*** if we want to perform the custom signs attack.

Than, if we want to perform the targeted attack, we can create a variable *tg* that must contain the label of the target class.

At this point we load our samples, the masks, in the *x_smp* and masks variables using the *load_samples* function, the path of the directory of the images will change according to the value of *attack_type*.

VII. Attacks evaluation

To evaluate our attacks, we created two pandas data frames (one for the evaluation of the fast gradient attack and the other for the evaluation of the iterative attack), with the following columns:

- **path_original:** the path of the original image;
- **prevision_original:** the prevision of the original image made by the model;
- **path_adversarial:** the path of the adversarial image created by the attack;
- **prevision_adv:** the prevision of the adversarial image made by the model;
- **success:** we have two cases:
  - **Untargeted attack:**
    - 1, if prevision_adv $\neq$ prevision_original;
    - 0, otherwise.
  - **Targeted attack:**
    - 1, if prevision_adv = target;
    - 0, otherwise.

At this point, each type of attack has its own folder to save its images and its own.*CSV* file, and the opened and read to compute the accuracy of the two attacks.

VIII. Detection phase

The detection phase is very important, since the network must classify images of the real world, with backgrounds, etc. So, this phase allows you to detect the area of interest for the model of the image, in our case a circle, clean it and send it to the model to classify it.

This phase is characterized by five functions, starting from the original image:

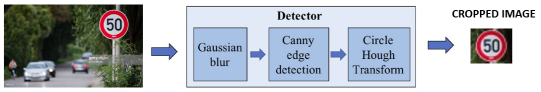

Figure 6: *Datection phase*

Gaussian Blur: This technique (Image Smoothing) help in reducing the noise. Any sharp edges in images are smoothed while minimizing too much blurring.

Gray Scale: This technique convert it to gray scale.

Canny Edge detection: This technique allows to easily detect the edges, distinguishing them in more or less relevant.

Circle Hough transform: is an image transform that allows for circular objects to be extracted from an image, even if the circle is incomplete. The transform effectively searches for objects with a high degree of radial symmetry, with each degree of symmetry receiving one "vote" in the search space. we specify the expected radius of interest and the quantity of circles to detect (in our case one). We set the *mg_ration* parameter to 0.4, to retrieve just the portion of the signal and to do take as less background as possible, and *n_circle* to 1, to look for exactly one circle (otherwise, it could find even the image portions that could be identified as circle, but with lower probability, incrementing the probability to have errors).

IX. Real time detection

In our project we have also implemented a phase of detection and classification in real time. In our test, for simplicity, we used the webcam of our pc, showing images of some road signs. The sign is detected in the camera frame and then classified by our model, showing the relative label above the frame that identifies the road sign.

We show an example in the following image:

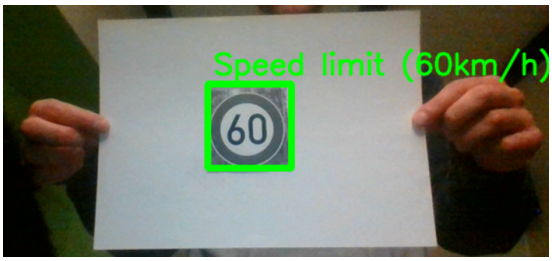

Figure 7: *Real time analysis*

Intuitively, this process can be generalized to other applications, for example, by mounting the camera in the dashboard of our car.

X. Result

We tested all kinds of attacks and, using as target class: we obtaining the following results (in % of accuracy):

- **In-distribution attack:**
  - Fast gradient attack:
    - Targeted:
    - Untargeted:
  - Iterative attack:
    - Targeted:
    - Untargeted:
- **Logo attack:**
  - Fast gradient attack:
    - Targeted:
  - Iterative attack:
    - Targeted:
- **Custom Signs attack:**
  - Fast gradient attack:
    - Targeted:
  - Iterative attack:
    - Targeted:

XI. Conclusions

Useful links

- **GitLab directory:**
  `github.com/leonardouniromasalvucci/DARTS-Deceiving-Autonomous-Cars-with-Toxic-Signs-`
- **Link to download the model:**
  `kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign/data`
- **Link to download the dataset:**
  `drive.google.com/open?id=1wHJzXTC_lVbN86NPu_hUpoaSxM9-v5m`
- **Link to the original paper:**
  `arxiv.org/pdf/1802.06430.pdf`

Contacts

- **Antonino Di Maggio's LinkedIn profile:**
  `linkedin.com/in/antonino-di-maggio/`
- **Leonardo Salvucci's LinkedIn profile:**
  `linkedin.com/in/leonardo-salvucci/`