Adversarial Attacks Against Machine Learning models In Automotive Industries

Mayank Chandnani *Pulkit Gambhir**
20860563 *20850709*
m2chandn@uwaterloo.ca *p4gambhi@uwaterloo.ca*
*both authors contributed equally

Abstract

Machine Learning and deep learning techniques are often associated with various intriguing ideas, one of those ideas is self-driving cars. Machine learning has yielded amazing advances in this particular field by training on numerous datasets. However, recent accidents that involve self-driving cars put a light on failure of neural networks, that were unable to recognize objects. Our project focuses on, creating algorithms which creates adversarial attacks on neural networks and tests the strength of these neural networks which are trained on GTSRB dataset. Four techniques were created and tested on three different models in the lifecycle of this project. In our first approach we created a random fuzzer that would modify the RGB values of the pixels randomly till it misclassifies the image. For our second approach, we applied a Gaussian filter on the images and checked for misclassification. In our third approach, we blended two images of the same class and then applied Gaussian filter on them. For our fourth and final approach we, implemented the FGSM method that adds noise based on gradient of the image. Of all the four methods, FGSM method gave us the most desirable output.

1. Introduction

As we know, there have been various advancements in the automotive industry. The advancements which include recognizing objects and traffic signal signs and accordingly, commanding the self-driving cars to drive, is a direct result of linking the field of machine learning and automobile industry. But there have been instances where self-driving cars have failed to recognize objects and resulted in a crash [1]. These instances are due to objects/traffic signals which were successful to fool the existing trained models.

The main objective of our project is to apply adversarial attacks on well trained machine learning models, in order to find out how many manipulations to an image does it take for a well-trained model to misclassify a traffic sign. Our aim is to keep the manipulations as minimal as possible such that the final image is recognizable by the human eye.

This topic in particular is a “hot” topic right now, as there are many companies, 55 alone in California which have the permit to make self-driving cars. There have been many failed attempts in making a perfect self-driving car which is impossible to fool. This project will help in the research, focusing on various parameters due to which recognizing a traffic signal using machine learning fails. Based on the results of the adversarial attack, one can make another dataset by manipulating the training images by tweaking, using the similar approaches that we used. By training on the new data set, the robustness of the models will improve. Thus, making it harder to fool a model.

The aim of the project was to develop algorithms which perform an adversarial attack on the neural networks chosen and compare them. We initially made a random fuzzer (naive approach) which would take an image and randomly choose the pixels of that image and change its RGB value to (255,255,255) i.e. white. This approach is inspired by One Pixel Attack for fooling deep neural networks. Our aim was to check the number of pixels required to break the strength of the network i.e. neural networks should predict wrong label. We observed that it took 280 pixels to break our neural network 1 and neural network 3 and it took 400 pixels to break the strength of neural network 2. But the resultant image was of a very bad quality.

This is the reason, this naive algorithm fails, because the model is predicting wrong labels after manipulation, but the quality of the image is deteriorated so drastically, that it is very difficult for the human eye to recognize the traffic sign.

The second algorithm applies a Gaussian Filter on the image. We observed that we must apply Gaussian Filter several times in order to break the model. After manipulating the image, we observed that image was recognizable by the human eye and the model was predicting the wrong label in few cases. This algorithm showed better results than random fuzzing as far as the image quality was concerned.

The third algorithm blends two images of the same class and then applies Gaussian Filter on them. After applying this algorithm on the images, we found that the images were of superior quality than that of images obtained after Random Fuzzing. But when we applied this particular algorithm on 200 images for our 3 neural networks, the accuracy score didn't go down significantly for neural networks 1 and 2. Thus, this algorithm is not successful for neural networks 1 and 2. The quality of the image is better than random fuzzing but it is not very effective in breaking the neural network, although it got the prediction accuracy of model 3 down to 66.5% from 96%.

Finally, our fourth algorithm Fast Gradient Sign Method (FGSM) got the prediction accuracy score down as well as the quality of the images are far superior than the other algorithms. The objective of this algorithm was to create adversarial attacks on neural network. We applied this algorithm with different values of epsilon and observed at which value, the model starts to predict wrong labels. The epsilon value found out is 0.1. This approach is highly successful as it gets the accuracy score low and also the image obtained is of good quality i.e. easily recognized by human eye.

Unfortunately, there were some compatibility issues with Tensorflow and Keras for our networks 1 and 2. Thus, we had to train our own neural network (model 3) and apply the FGSM algorithm on it. We however, applied the other three algorithms on model three as well to maintain uniformity for comparison.

2. Description of Algorithm

2.1 Random Fuzzer

The aim of this algorithm is to simply change random RGB values of the pixels and see how many pixels it takes for the neural network to misclassify the image. We took two approaches to implement this logic, one for single images and one for set of images.

For our first approach on a single image, we first determined the size of the image. Then, depending upon the size of the image, random pixels were selected from the image. Once we had the random pixels with us, we modified the RGB values of these pixels by 10. We check the class of the image again to see if it has been misclassified or not. If not, we pick another set of random pixels and modify their values. We keep this going in an iterative loop until the image is misclassified. Fig. 1 shows the progress of an image as pixels are modified using this approach. Code 1 is the pseudocode for this approach and Code Snippet 1 is a snippet from the actual implementation of the code. The images shown in Fig.1 and Fig.2 aren't the final images after misclassification, those images have been later covered in Section 4 of this report.

```
1. loop{misclassification}{
2.   for pixel in imageSize/2
3.        create list_of_random_pixels
4.
5.   for pixel in list_of_random_pixels
6.       (R,G,B) of pixel = (R+10, G+10, B+10) of pixel
7.  }
```

Code 1: Pseudocode for Approach 1

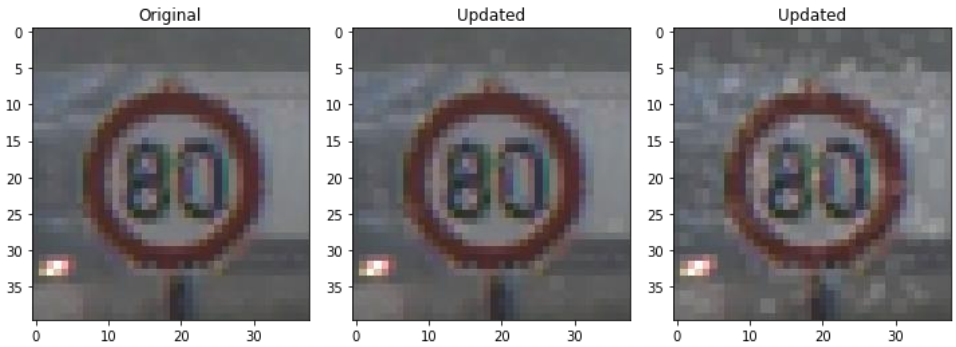

Fig. 1 Image progress for Approach 1

```
for i in range(0,int(xSize/2),1):
    li_x.append(random.randint(0,xSize))
    li_y.append(random.randint(0,ySize))
for i in range(0,int(xSize/2),1):

        p=255 if img[li_x[i],li_y[i]][0]>245 else img[li_x[i],li_y[i]][0]+10
        q=255 if img[li_x[i],li_y[i]][1]>245 else img[li_x[i],li_y[i]][1]+10
        r=255 if img[li_x[i],li_y[i]][2]>245 else img[li_x[i],li_y[i]][2]+10
        img[li_x[i],li_y[i]]= (p,q,r)
```

Code Snippet 1: Code for Approach 1

In our second approach for a set of images, we modified the randomly selected pixels to the value of (255,255,255). This was done, because in approach 1 it was observed that for different images, different amount of iterations were needed to obtain the result, the iterations varied from 50 to 1200 and it was computationally costly to run the loop for multiple iterations for each image in a large set of images. Thus, we decided to choose a specific number of pixels (by observing the breaking points on few images), randomly select their coordinates of the images and change the values to extreme values so that the number of iterations reduce. This however came at the cost of modified image deviating from original image visually. Fig. 2 showcases the progress of image in approach 2. Code 2 is the pseudocode for this approach and Code Snippet 2 is a snippet from the actual implementation of the code.

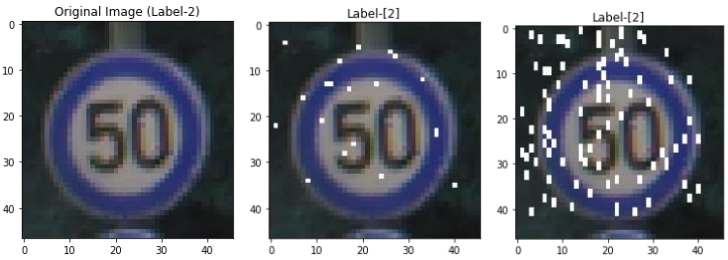

Fig.2 Image progress for Approach 2

```
1. loop{misclassification}{
2.   for pixel in imageSize/2
3.        create list_of_random_pixels
4.
5.   for pixel in list_of_random_pixels
6.       (R,G,B) of pixel = (255,255,255) of pixel
7.  }
```

Code 2: Pseudocode for Approach 2

```
for i in range(0,int(xSize/2),1):
    li_x.append(random.randint(0,xSize))
    li_y.append(random.randint(0,ySize))
for i in range(0,int(xSize/2),1):
    img[li_x[i]:li_x[i]+1,li_y[i]:li_y[i]+1] = (255,255,255)
```

Code Snippet 2: Code for Approach 2

2.2 Gaussian Filtering

In this algorithm we applied Gaussian Filter. The kernel of height 5 and width 5 is convolved over the whole image matrix covering the pixels. It blurs the image by prioritizing the pixel of interest. In order to apply a filter on a specific pixel, a weighted average of the colour values of the pixels in the kernel is calculated. The pixel of interest (pixel which is to be filtered) is always the centre of the kernel. As we know the shape of the Gaussian Curve is similar to a bell curve which means the maximum weight is in the middle compared to those which are far away from centre. The kernel is made up of floating points with maximum weight in the middle to edges of the matrix. Edges are preserved in this kind of blur when compared to mean blurring.[7]

Fig. 3 [6] depicts a 2D Gaussian Mask (Kernel) being multiplied by the image matrix. For this figure, k0 will have the maximum weight compared to the sides and X5 will be replaces by PC which will be equal to -:

PC= (K1*X1+K2*X2+K3*X3+K2*X4+K0*X5+K2*X6+K1*X7+K2*X8+K1*X9)/9

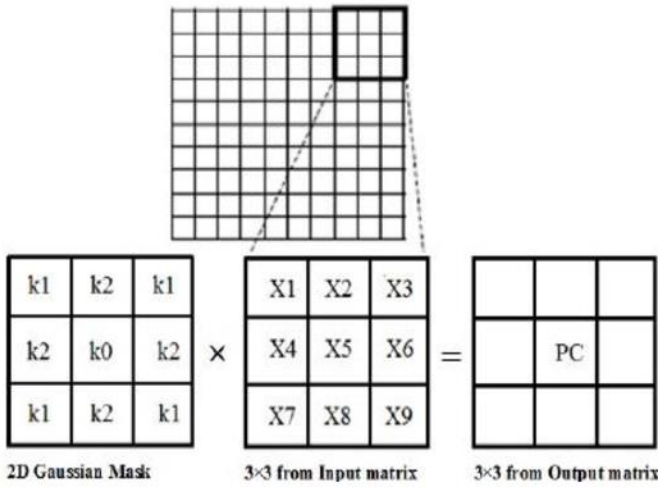

Fig. 3: Convolution Operation

```
1. load image;
2. loop{3}{
3.   apply Gaussian filter on image
4.  }
```

Code 3: Pseudocode for Method 2

```
for i in range(0,3):
    img = cv2.GaussianBlur(img,(5,5),0)
```

Code Snippet 3: Code snippet for Method 2

Code 3 is the pseudocode for method 2 and code snippet 3 is the snippet from the actual implementation of the code.

2.3 Combination of Blending and Gaussian Filtering

Blending is the process of combining two images together. The objective of this algorithm is to blend two images of the same labels from the GTSRB dataset and then apply Gaussian blur over it. As the dataset for each class in GTSRB is very vast, the images were chosen such that they are images of two varying qualities.[8]

The blending is done using the following formula-:

blended image= alpha * Image 1 + beta * Image 2 + gamma

where, Image 1- First image in array, alpha- weight of the first array elements, Image 2- Second image in array of the same size and channel as Image1, beta- weight of the second array elements, gamma- scaler added to each sum

The values of alpha, beta and gamma were chosen after trial and error. After a blended image is obtained, Gaussian Blur is applied on that image. The pseudo code for fetching a random image, loading the model, predicting the label for this algorithm is similar to our second algorithm. The only difference is that in order to applying combination of blending and Gaussian Filtering.

```
1. load image_1;
2. load image_2;
3. blend Gaussian filter on image
4. apply Gaussian filter on image
```

Code 4: Pseudocode for Method 3

```
alpha=0.5
beta=0.3
gamma=0.2
output=cv2.addWeighted(img_1,alpha,img_2,beta,gamma)  #blen
output_gauss = cv2.GaussianBlur(output,(5,5),0)       # app
```

Code Snippet 4: Code Snippet for Method

2.4 Fast Gradient Sign Method

In this method we add noise to the original image. The noise is added based on the gradient of the loss function with respect to the input image. The noise thus obtained is added to the original image in the direction of loss function.

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

Where, adv_x : Adversarial image, x : Original input image, y : Original input label, ε: Multiplier to ensure the perturbations are small. θ: Model parameters. *J*: Loss.

The noise is scaled by epsilon, which is usually constrained to be a small number via max norm.

```
1. load image;
2. predict output;
3. find loss_function based on actual class and predicted class
4. find gradient of loss_function w.r.t input image
5. multiply gradient with epsilon
6. add disturbance to original image
```

Code 4: Pseudocode for Method 4

```
def create_adversarial_pattern(input_image, input_label):
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        prediction = pretrained_model(input_image)
        loss = loss_object(input_label, prediction)

    # Get the gradients of the loss w.r.t to the input image.
    gradient = tape.gradient(loss, input_image)
    print(gradient)
    # Get the sign of the gradients to create the perturbation
    signed_grad = tf.sign(gradient)
    return signed_grad
```

Code Snippet 4: Implementation of method 4

3. Result

3.1. Model and dataset description

All the three models that we are using have been trained on the German Traffic Sign Recognition Benchmark (GTSRB) dataset [2]. The dataset consists of more than 50000 images that have been classified into 43 different classes. The dataset is pretty exhaustive with each class having images taken in different lighting and environmental settings.

The first model was developed by user [3] and has an accuracy of 99%. It selected uses 4 convolutional layers with activation function ReLU (Rectified Linear Unit), two pooling layers and 2 dense layers with the output layer using Softmax activation.

The second model was developed by user [4] and has an accuracy of 95%. It uses 5 convolutional layers with 2 using ReLU activation function, 2 using ELU (Exponential Linear Unit) activation function and one using Sigmoid activation function.

The third model was developed by us [5][12] and has an accuracy of 96%. It uses 5 convolutional layers with ReLU as the activation function, two pooling layers, and two dense layers with one layer using ReLU and the output layer using Softmax.

3.2 Random Fuzzing

In **random fuzzing** as mentioned above, two approaches were used. In approach 1, the result obtained when varying the pixels until breakpoint varied a lot. The number of iterations required for an image to misclassify varied between 50 to 1200 for some cases. Not only were the required iterations varying but the image output also varied based on the number of iterations.

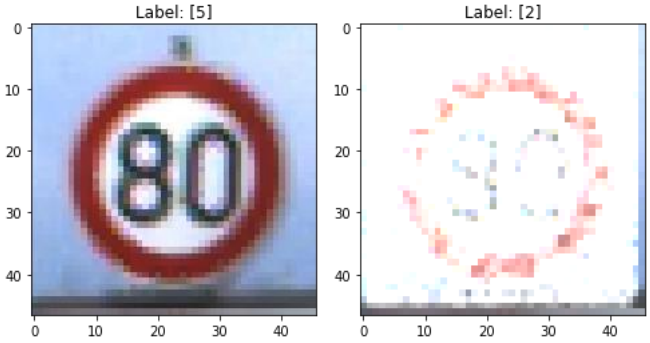

Fig.4: Misclassification after 422 loops

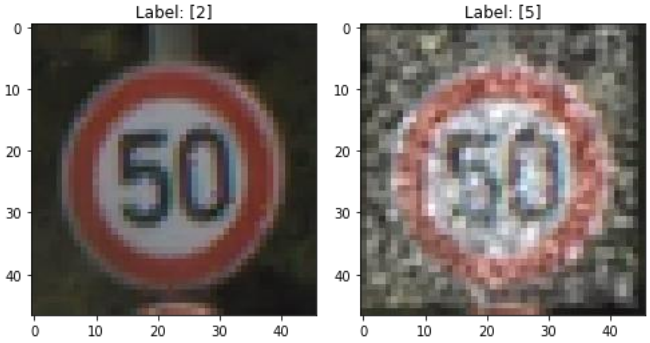

Fig.5: Misclassification after 130 loops

Fig. 4 and Fig. 5 depict the variation in random fuzzing output. For fig.4, it took 422 iterations to misclassify the image and the output image looked very different from the original image. Meanwhile, the same fuzzer misclassified an image in 130 iterations and the final image wasn't that different from the original image. Here, when we say one iteration, the process of selecting random pixels and modifying them is considered as one iteration. Thus, this method is very unreliable.

In random fuzzing, approach 2, we changed the values of the pixels to (255,255,255) directly. This approach was more suited for batch images as it needed fewer iterations but the final image after this approach was drastically different from the original image.

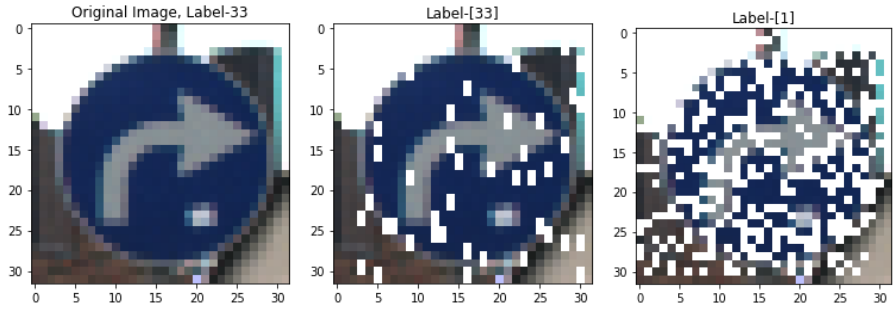

Fig 6: Misclassification using approach 2

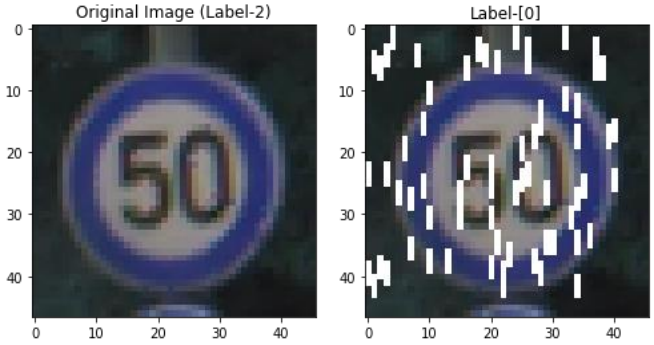

Fig 7: Misclassification using approach 2

As it can be seen from Fig 6 and Fig 7, the final image obtained after applying this approach are very different from the original image. Moreover, it can also be observed that the amount of perturbation is very different for both the examples is very different, thus this approach is also unreliable.

3.3 Gaussian Filter

In method 2 we applied **Gaussian Filter** with kernel of size 5X5. We observed for one clear image, that it took 3 iterations for the Gaussian Filter with height 5 and width 5 to transform an image, such that the model fails to classify it properly.

Fig. 8 depicts the effect that Gaussian filter have on images; we applied a Gaussian filter thrice to the images. After applying it more than thrice, the images were deviating a lot from the original image.

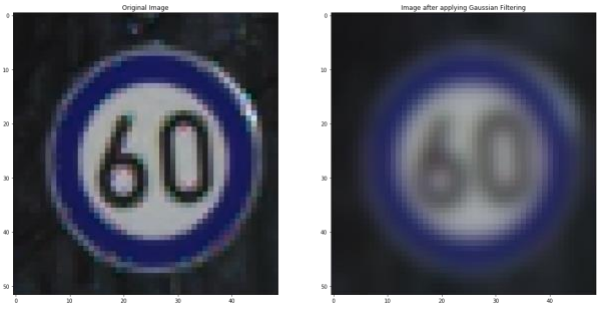

Fig 8: Gaussian Filter

As it is clear from Fig 8, the image obtained after applying Gaussian filter isn't as drastically different from the original image. However, as it is clear from Table 1, this approach is not able to decrease the accuracy much. Thus, though this approach maintains the original image, it loses out on accuracy.

3.4 Blending and Gaussian filter

The **blending and Gaussian filter** method adds on to the previous one by blending two images and then applying Gaussian filter. As it can be seen from Fig 9 and Table 1, this doesn't change the image much, but the accuracy isn't affected much either.

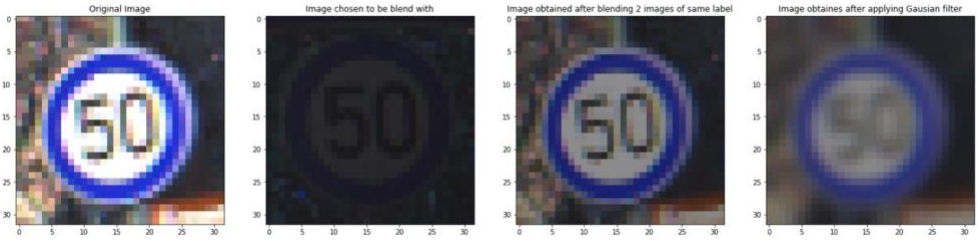

Fig 9: Blending followed by Gaussian filter

3.5 FGSM method

The **FGSM** method is by far the best method amongst all methods. The value for epsilon was calculated to be 0.1 after trial and error over images. The images don't change much after applying this approach and the accuracy drops drastically.

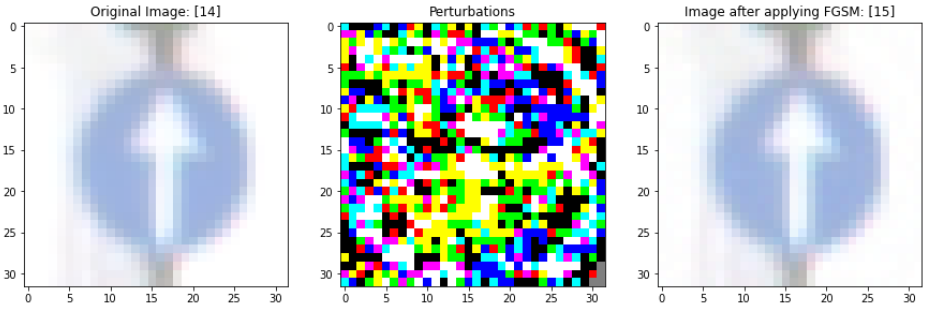

Fig 10: FGSM process

As it can be seen from Fig 10, the image is misclassified after adding the disturbance and the final image isn't that different from the original image.

Table 1 and Fig. 11 show the final accuracies obtained after implementing the layers.

Model	Actual	Random	Gaussian	Blending+Gaussian	FGSM
Model 1	99%	54%	87.5%	93%	-
Model 2	95.5%	46%	78.5%	90%	-
Model 3	96%	44%	58.5%	66.5%	2.5%

Table 1: Accuracies on 200 images

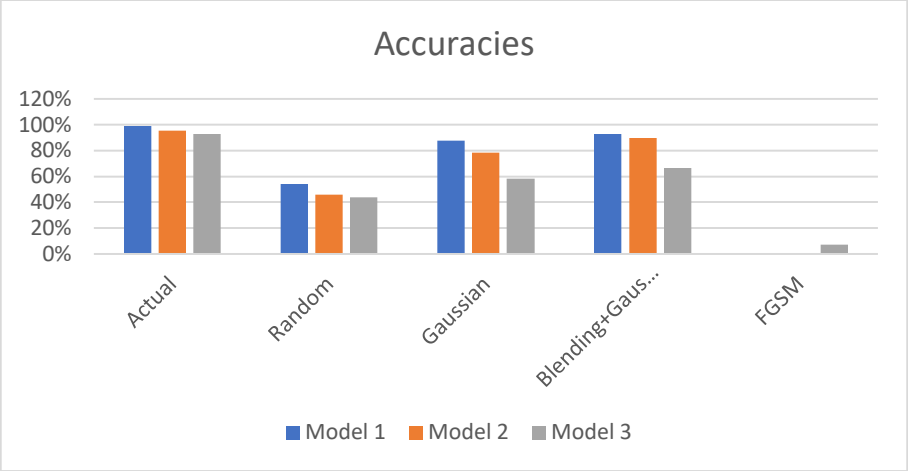

Fig 11: Accuracy Chart

4. Conclusions

It is clear from the above-mentioned results, that FGSM is the best approach among the four approaches displayed here.

The project can be further developed by solving the Tensorflow and Keras compatibility issues that were faced by us while implementing FGSM for models 1 and 2. Once the images have been obtained after applying FGSM filters on it, the new images can be used to train the models again. This can be done only with the images obtained after applying FGSM and not the other methods shown in this project because the image doesn't change much in FGSM.

Furthermore, various toolboxes implementing various adversarial attacks are available online and can be explored. One such toolbox is Foolbox [9] which can be used to implement attacks like FGSM, Inversion attacks, boundary attacks, etc. Another such toolbox is CleverHans [10] which is equipped to implement attacks like FGSM, Jacobian based saliency attacks and various black box attacks. Lastly, white box testing frameworks like DeepXplore [11] can also be used as further research benchmarks for this project.

References

[1] https://www.cbc.ca/news/business/tesla-s-self-driving-autopilot-system-under-scrutiny-1.5413931
[2] https://www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign
[3] https://github.com/ItsCosmas/Traffic-Sign-Classification
[4] https://github.com/ulmefors/Traffic-Sign-Classifier
[5] https://git.uwaterloo.ca/m2chandn/model_3_ece653/-/tree/master
[6] https://www.semanticscholar.org/paper/Reducing-the-Computational-Complexity-Of-A-2d-for-Raj-Singal/563f7beebee994d250da696236d524e64584e9e6
[7]https://datacarpentry.org/image-processing/06-blurring/
[8]https://docs.opencv.org/3.4/d2/de8/group__core__array.html#gafafb2513349db3bcff51f54ee5592a19
[9]https://foolbox.readthedocs.io/en/stable/modules/attacks.html#foolbox.attacks.BoundaryAttack
[10] https://github.com/tensorflow/cleverhans
[11] https://github.com/peikexin9/deepxplore
[12] https://towardsdatascience.com/building-a-road-sign-classifier-in-keras-764df99fdd6a