


Table of contents

MLOps Blog

Adversarial Attacks on Neural Networks: Exploring the Fast Gradient Sign Method

 5 min

 Henry Ansah

 21st April, 2023

ML Model Development

Since their invention, [neural networks](#) have always been the crème de la crème of machine learning algorithms. They have driven most of the breakthroughs in artificial intelligence.

Neural networks have proven to be robust at performing highly complex tasks that even humans find very challenging.

Can their incredible robustness extend beyond their original purpose? That's what we'll try to find out in this article.

Personally, one area that I never expected to intersect with AI is **security**. As it turns out, this is one of the few areas where neural networks fail.

We'll try a very popular attack, the [Fast Gradient Sign Method](#), to demonstrate the security vulnerabilities of neural networks. But first, let's explore different categories of attacks.

Adversarial attacks

Table of contents

Depending on what you as the attacker know about the model you wish to fool, there are several categories of attacks. The two most popular are the **white box attack** and **black box attack**.

Both of these attack categories have the general goal of fooling a neural network into making wrong predictions. They do it by adding hard-to-notice noise to the input of the network.

What makes these two types of attacks different is your ability to gain access to the entire architecture of the model. With white box attacks, you have complete access to the architecture (weights), and the input and output of the model.

Control over the model is lower with black box attacks, as you only have access to the input and output of the model.

There are some goals that you might have in mind when performing either of these attacks:

- **misclassification**, where you only want to drive the model into making wrong predictions without worrying about the class of the prediction,
- **source / target misclassification**, where your intention is to add noise to the image to push the model to predict a specific class.

The Fast Gradient Sign Method (FGSM) combines a white box approach with a misclassification goal. It tricks a neural network model into making wrong predictions.

Let's see how FGSM works.

Fast Gradient Sign Method explanation

The name makes it seem like a difficult thing to understand, but the FGSM attack is incredibly simple. It involves three steps in this order:

1. Calculate the loss after forward propagation,
2. Calculate the gradient with respect to the pixels of the image,
3. Nudge the pixels of the image ever so slightly in the direction of the calculated gradients that maximize the loss calculated above.

Table of contents

The first step, calculating the loss after forward propagation, is very common in machine learning projects. We use a negative likelihood loss function to estimate how close the prediction of our model is to the actual class.

What is not common, is the calculation of the gradients with respect to the pixels of the image. When it comes to training neural networks, gradients are how you determine the direction in which to nudge your weights to *reduce* the loss value.

Instead of doing that, here we adjust the input image pixels in the direction of the gradients to *maximize* the loss value.

When training neural networks, the most popular way of determining the direction in which to adjust a particular weight deep in the network (that is, the gradient of the loss function with respect to that particular weight) is by back-propagating the gradients from the start (output part) to the weight.

The same concept applies here. We back-propagate the gradients from the output layer to the input image.

In neural network training, in order to nudge the weights to decrease the loss

Play with a live Neptune project -> Take a tour 



neptune.ai

$new_weights = old_weights - learning_rate * gradients$



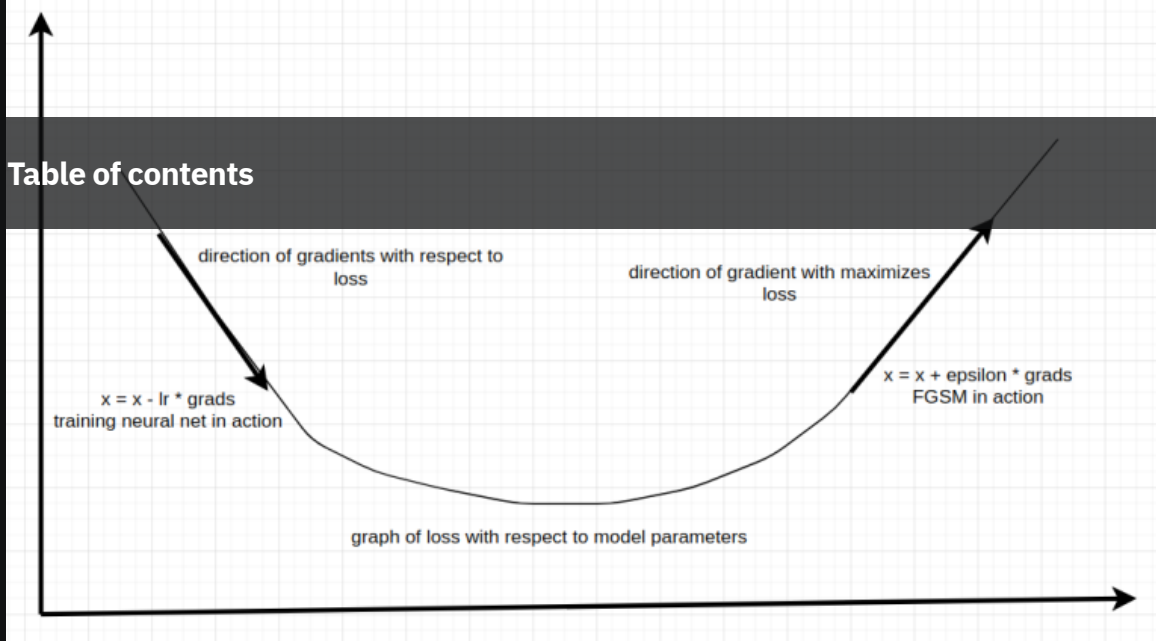
Blog



ML Model Development

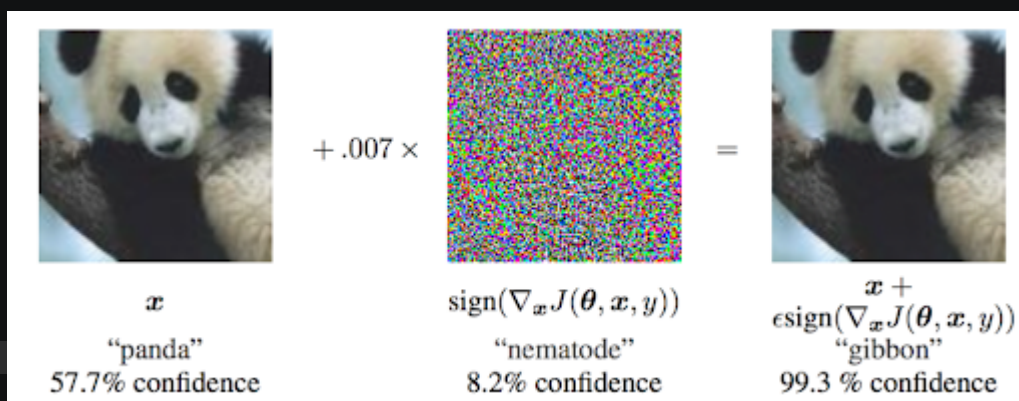


$new_pixels = old_pixels + epsilon * gradients$



In the image above, we see two arrows representing two different ways of adjusting gradients to achieve a goal. The equation on the left hand side, as you might have probably guessed correctly, is the fundamental equation involved in training neural networks. Naturally, the gradients computed point to the direction that maximizes the loss. The negative sign in the **neural networking training equation** ensures that the gradient points in the opposite direction – the direction that minimizes the loss. That is not the case with the equation on the right hand side which is the equation involved in fooling the neural network. Since we want to maximize the loss, we apply the gradients in their natural form, so to speak.

There are a number of differences between the two equations. The most important one is addition and subtraction. By using the structure of equation two, we nudge the pixels in the direction opposite to the direction that minimizes the loss. By doing so, we are telling our model to do just one thing – make wrong predictions!



Source

In the image above, x represents the input image that we want the model to predict. The second part of the image represents the gradients of the loss function with respect to the input image.

Remember that the gradient is just a directional tensor (it gives information about which direction to move in). In order to enforce the nudging effect, we multiply the gradients with a very tiny value, epsilon (0.007 in the image). Then we add the result to the input image, and that's it!

The alarming expression under the resulting image can simply be put this way:

$$\text{input_image_pixels} + \text{epsilon} * \text{gradient of loss function with respect to the input_image_pixels}$$

At this point, let's summarize what we've talked about so far, and next we'll do some coding. In order to trick a neural network into making wrong predictions, we:

- forward-propagate our image through our neural network,
- calculate the loss,
- back-propagate the gradients to the image,
- nudge the pixels of the image in the direction that maximizes the loss value.

By doing so, we tell the neural network to make a prediction about the image that is far from the correct class.

One thing we need to note is that the degree of noticeability of the noise on the resulting image depends on the **epsilon** – the larger the value, the more noticeable the noise.

Increasing epsilon also increases the possibility of the network making a wrong prediction.

The code

In this tutorial, we're going to use TensorFlow to build our entire pipeline. We'll focus on the most important parts of the code, without going into parts related to data processing.

First, we're going to load one of TensorFlow's state-of-the-art models, the **MobileNet V2 model**:

```
pretrained_model =
tf.keras.applications.MobileNetV2(include_top=True,

weights='imagenet')
pretrained_model.trainable = False
```

Setting the model's trainable property to *false* means our model cannot be trained hence any effort to change the model parameters in order to fool the model is going to fail.

We may want to visualize the images to get a sense of how different values of **epsilon** affects the predictions as well as the nature of the image. The simple snippet of code takes care of that.

```
def display_images(image, info):
    _, label, prob =
get_imagenet_label(pretrained_model.predict(image))
    plt.figure()
    plt.imshow(image[0]*0.5+0.5)
    plt.title('{} n {} : {:.2f}%
prob.format(info, label, prob*100))
    plt.show()
```

Next, we load our image, run it through our model and obtain the gradients of the loss with respect to the image.

```
loss_object = tf.keras.losses.Categorical_Crossentropy()
def create_adversarial_pattern(input_image, input_label):
    with tf.GradientTape() as tape:
        tape.watch(input_image)

        #forward propagate image and retrieve the loss
```

```
prediction = pretrained_model(input_image)
loss = loss_object(input_label, prediction)
```

```
# obtain the gradient of the loss with respect to the
```

Table of contents

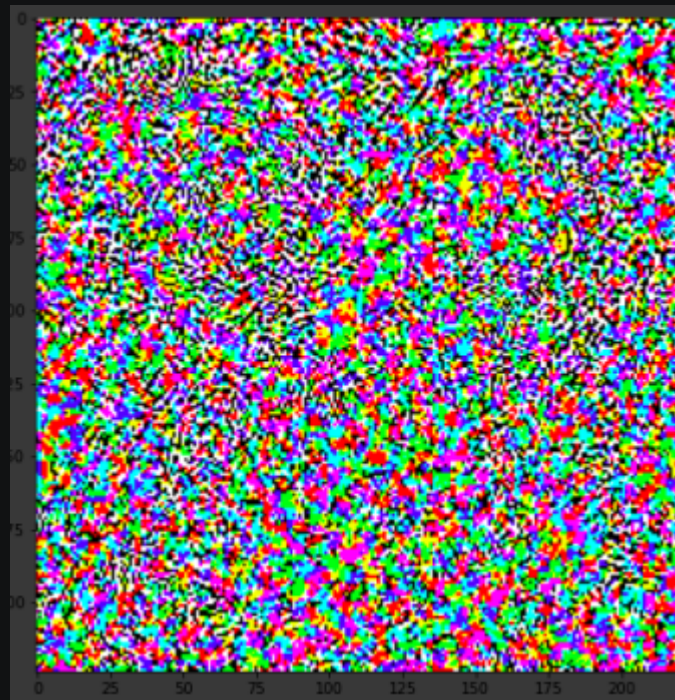
```
gradient = tape.gradient(loss, input_image)
```

```
# Get the sign of the gradients (directions)
```

```
signed_grad = tf.sign(gradient)
```

```
return signed_grad
```

Printing signed_grad shows a tensor of ones. Some with a positive sign, others with a negative sign, which indicates that the gradients only enforce a directional effect on the image. Plotting it reveals the image below.



Now that we have the gradients, we can nudge the image pixels in the direction opposite to the direction of the gradients.

In other words, we nudge the image pixels in the directions that maximize the loss. We are going to run the attack on different values of epsilon, with **epsilon=0** indicating no attack is being run.

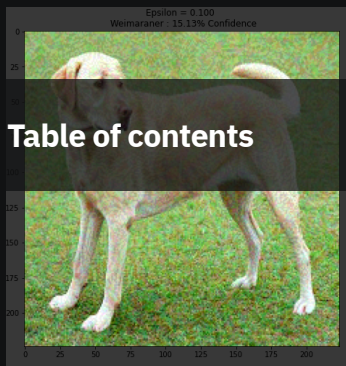
```
epsilons = [0, 0.01, 0.1, 0.15]
descriptions = [('Epsilon = {:.3f}'.format(eps) if eps else
```



```
'Input')  
  
        for eps in epsilons]  
    for i, eps in enumerate(epsilons):  
        adv_x = image + eps*perturbations #input_image + epsilon *  
  
Table of contents  
    adv_x = tf.clip_by_value(adv_x, -1, 1)  
    display_images(adv_x, descriptions[i])
```

We get the following results:





Notice the pattern in the three images above? With increasing value of epsilon, the noise becomes more visible, and the confidence for a wrong prediction increases.

We have successfully fooled a state-of-the-art model into making wrong predictions, without changing anything about the model.

Let's perform a little experiment here to confirm a concept we discussed above. Instead of adding the result of the tensor multiplication of epsilon and the gradients (image + eps * signed_grad) to the image, which nudges the pixels of the image in the direction of maximizing the loss, we're going to perform subtraction, which nudges the pixels of the image in the direction which minimizes the loss (image - eps * signed_grad).

```

epsilons = [0, 0.01, 0.1, 0.15]
descriptions = [('Epsilon = {:.3f}'.format(eps) if eps else
                'Input')
                for eps in epsilons]
for i, eps in enumerate(epsilons):
    adv_x = image - eps*perturbations #input_image - epsilon *
    gradients
    adv_x = tf.clip_by_value(adv_x, -1, 1)
    display_images(adv_x, descriptions[i])

```

The result:

Table of contents



By nudging the image pixels in the direction of the gradient that minimizes the loss, we increase the confidence of the model in making correct predictions much more than without this experiment. It went from 41.82% confidence to 97.89%.

Next steps

Since the invention of the FGSM, several other methods were created with different angles of attacks. You can check out some of these attacks here: [Survey on Attacks](#).

Table of contents out different models and a different image. You may also build your own model from scratch and also try out different values of epsilon.

That's it for now, thank you for reading!



Henry Ansah

Student

Follow me on

Read next

The Advantages of Synthetic Data Over Real Data

Artificial intelligence is all the rage in 2020, but many aspiring technologists are running into a problem: training data.

Having a large, curated dataset is necessary for most artificial intelligence/machine learning applications. Acquiring that data is often a challenge.

Table of contents If you have to collect data from the real world, you must annotate and prepare it for your model. For students, small research teams, and early-stage startups, training data is a significant hurdle to overcome.

That's where synthetic training data comes in handy. Synthetic data is fake data that mimics real data.

For certain ML applications, it's easier to create synthetic data than to collect and annotate real data.

There are three major reasons for this:

you can generate as much synthetic data as you need, you can generate data that may be dangerous to collect in reality, synthetic data is automatically annotated.

Let's get into the details.

What is synthetic data?

One of the fundamental laws of machine learning is that you need a lot of data.

The amount of data ...

[Continue reading](#)

Table of contents

How to Build ML Model Training Pipeline

by Henrique Pett, 10 min read

[Read more](#)

Table of contents

Building ML Platform in Retail and eCommerce

by Shibsankar Das, 10 min read

[Read more](#)



Table of contents

Top MLOps articles, case studies, events (and more) in your inbox every month.

[Get Newsletter](#)

PRODUCT



DOCUMENTATION



COMPARE



COMMUNITY



COMPANY



[The Best MLOps Tools](#) • [MLOps at a Reasonable Scale](#) • [ML Metadata Store](#) •
[MLOps: What, Why, and How](#) • [Experiment Tracking in Machine Learning](#)

[Terms of service](#) [Privacy policy](#)

Copyright © 2022 Neptune Labs. All rights reserved.