Lab : Carlini-Wagner Attack

1 Introduction

1.1 Carlini-Wagner Attack (CW-l2):

The Carlini-Wagner attack is a significant method in adversarial machine learning aimed at generating adversarial examples that can mislead neural networks. The attack is based on finding small perturbations to input data that cause misclassification while keeping the changes imperceptible. Carlini and Wagner's method is highly effective against networks, even those with some defenses like defensive distillation.

The research focuses on optimizing the adversarial examples with respect to different distance metrics, such as the L_2 norm, L_0 norm, and L_∞ norm, which measure the level of perturbation. These metrics help generate adversarial examples that evade detection while remaining close to the original input data. This method has been used extensively to test the robustness of models used in image classification tasks.

In more detail:
1. L_0 distance measures the number of coordinates i such that $x_i \neq x'_i$. Thus, the L_0 distance corresponds to the number of pixels that have been altered in an image.

2. L_2 distance measures the standard Euclidean (root-mean-square) distance between x and x'. The L_2 distance can remain small when there are many small changes to many pixels.

3. L_∞ distance measures the maximum change to any of the coordinates:

$$\|x - x'\|_\infty = \max\left(|x_1 - x'_1|, \ldots, |x_n - x'_n|\right)$$

1.2 Loss Function in Carlini-Wagner Attack

The Carlini-Wagner attack optimizes a specific loss function that balances the trade-off between the perturbation size and the model's misclassification. The loss function generally has two components:

Misclassification Loss: This term ensures that the perturbed image is classified incorrectly by the target model. It typically involves adding a term that penalizes correct classifications and encourages misclassifications.

Perturbation Loss: This term controls the magnitude of the perturbation, and it is based on the chosen norm (e.g., L_2, L_∞, or L_0). It ensures that the perturbation is minimized according to the specific norm being used.

The general form of the loss function for the Carlini-Wagner attack is:

$$\text{Loss} = \text{Misclassification Loss} + \lambda \cdot \text{Perturbation Loss}$$

where λ is a trade-off parameter that balances the importance of the perturbation size versus the misclassification.

2 Environment Setup

2.1 Google Colab

Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.
To setup the Google colab:

1. Go to Google Colab using your web browser.
2. Sign in to your Google account .
3. Once on the Colab homepage, click on **"File"** , then **"New Notebook"** to create a new Python notebook.

4. Rename your notebook by clicking on the default name (e.g., "Untitled.ipynb") at the top and enter desired name.
5. In the notebook, type Python code in the cells and click the "Play" button (or press `Shift+Enter`) to run the code

3 Run Carlini-Wagner attack experiment

3.1 Install Cleverhans

First, you'll need to install the **cleverhans** library, which provides tools to benchmark the vulnerability of machine learning models to adversarial examples. Open a new code cell in your Colab notebook and run:

```
!pip install cleverhans
```

3.2 Libraries setup

```
import torch
import torchvision.transforms as transforms
import torchvision.models as models
from PIL import Image
import matplotlib.pyplot as plt
```

3.3 Integrating a Pretrained ResNet-18 Model for Image Classification

- **Load ResNet-18 Model**: `model = models.resnet18(pretrained=True)`
  Initializes ResNet-18 with pre-trained weights.
- **Set to Evaluation Mode**: `model.eval()`
  Configures the model for inference by disabling dropout and batch normalization updates.

```
model = models.resnet18(pretrained=True)
model.eval()
```

3.4 Transferring the Model to GPU (if available)

To make things run faster, especially with large models, we want to use the GPU if one is available. The code checks if a CUDA-compatible GPU is present, and if it is, we send the model to the GPU. If there's no GPU, the model will stay on the CPU. This ensures we're using the best hardware available for faster computations.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

3.5 Defining Image Transformations (Without Normalization)

To get images ready for the model, we need to apply a few transformations to make sure they're the right size and format. Here, we're setting up these transformations without normalization. First, we resize the image to 256 pixels, then crop it to a 224x224 square. Finally, we convert the image into a tensor, which changes the pixel values from the usual [0, 255] range to [0.0, 1.0]. This is a standard step for many deep learning models to ensure that all images are processed in a consistent way before being used.

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),  # Converts [0, 255] range to [0.0, 1.0]
])
```

3.6 Loading and Preprocessing the Image

To prepare an image for model inference, we start by loading it from the specified file path. After loading, we apply the previously defined transformations to resize, crop, and convert the image into a tensor. The unsqueeze(0) operation adds a batch dimension, which is required for processing by the model. Finally, the tensor is moved to the appropriate device (GPU or CPU) for efficient computation. This ensures the image is correctly formatted and ready for the model to analyze.

```
img = Image.open('/content/Horse.jpg')
input_tensor = transform(img).unsqueeze(0).to(device)
])
```

3.7 Defining the Normalization Function and Model Prediction

First, we create a normalization function to adjust the image in the same way the model was trained. This function ensures that our image data has the same mean and standard deviation as the data used during training. It's crucial for accurate predictions because it aligns with the model's expectations.
Next, we use this normalization function to get predictions from the model for our original image. By using `torch.no_grad()`, we avoid unnecessary calculations related to gradients, making the process faster and more efficient. We then print out the predicted class for the image.

We also set up a `model_fn` function, which normalizes the input image and then passes it through the model to get the output logits. This function is useful for evaluating the model's performance or for tasks that require this specific model function.

```
def normalize(x):
    mean = torch.tensor([0.485, 0.456, 0.406]).to(device).view(1, 3, 1, 1)
    std = torch.tensor([0.229, 0.224, 0.225]).to(device).view(1, 3, 1, 1)
    return (x - mean) / std

with torch.no_grad():
    output = model(normalize(input_tensor))
    _, pred_label = torch.max(output, 1)
    print(f'Original Prediction: {pred_label.item()}')

def model_fn(x):
    x_norm = normalize(x)
    logits = model(x_norm)
    return logits
```

3.8 Applying the Carlini-Wagner L2 Attack

Here, we start by importing the `carlini_wagner_l2` function from the CleverHans library, which will help us generate adversarial examples to challenge the model. We then use this function to create an adversarial version of our input image. The attack parameters include settings like the learning rate and the number of iterations, which guide how the attack modifies the image.

Once we have our adversarial image, we use the model to make a prediction on it, just as we did with the original image. This helps us see how the attack has affected the model's prediction.

Finally, we convert the adversarial tensor back to an image and save it as a file, so we can review and analyze how the attack has altered the image. A confirmation message is printed to let us know that the image has been saved successfully.

```
from cleverhans.torch.attacks.carlini_wagner_l2 import carlini_wagner_l2

adv_tensor = carlini_wagner_l2(
    model_fn=model_fn,
    x=input_tensor,
    n_classes=1000,
    y=pred_label,
    targeted=False,
    lr=0.01,
    confidence=0,
    clip_min=0.0,
    clip_max=1.0,
    initial_const=0.001,
    binary_search_steps=10,
    max_iterations=1000,
)

with torch.no_grad():
    adv_output = model(normalize(adv_tensor))
    _, adv_pred_label = torch.max(adv_output, 1)
    print(f'Adversarial Prediction: {adv_pred_label.item()}')

adv_img = transforms.ToPILImage()(adv_tensor.squeeze(0).cpu())
adv_img.save('adv_sample.jpg')
print('Adversarial image saved as adv_sample.jpg')
```

3.9 Displaying Original and Adversarial Images

Here, we start by converting our image tensors into PIL images so we can visualize them. This makes it easier to compare the original image with the adversarial one.
We then create a side-by-side view of both images using Matplotlib. Each image is displayed in its own subplot, with a title that shows the image along with the model's prediction for each one.

Finally, we use plt.show() to bring up the side-by-side images, letting us directly compare how the adversarial attack has changed the original image and how it affects the model's predictions.

```
original_img = transforms.ToPILImage()(input_tensor.squeeze(0).cpu())
adv_img = transforms.ToPILImage()(adv_tensor.squeeze(0).cpu())

fig, axs = plt.subplots(1, 2, figsize=(10, 5))  # Create 1 row, 2 columns of images

axs[0].imshow(original_img)
axs[0].axis('off')
axs[0].set_title(f'Original Image\nPrediction: {pred_label.item()}')

axs[1].imshow(adv_img)
axs[1].axis('off')
axs[1].set_title(f'Adversarial Image\nPrediction: {adv_pred_label.item()}')

plt.show()
```

3.10 Result

Figure 1: Original Image vs Adversarial Image

3.11 References

[1] Carlini, Nicholas, and David Wagner. "Towards evaluating the robustness of neural networks." 2017 ieee symposium on security and privacy (sp). Ieee, 2017.

[2] Biswas, I., Heu, V., & Hurtubise, J. (2016). Isomonodromic deformations of irregular connections and stability of bundles. arXiv preprint arXiv:1608.00780.

[3] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

[4] https://github.com/cleverhans-lab/cleverhans

[5] https://github.com/pytorch/pytorch