Finding the Best Attacks

Box-constrained L-BFGS

ALBERTO HOJEL[1], RYAN TABRIZI[1], AND HEATHER DING[1]

[1] *UC Berkeley, equal contribution*

Compiled August 4, 2023

This paper focuses on adversarial machine learning, examining the susceptibilities of neural network models and the development of robust defenses against adversarial attacks. The paper extends on previous work by implementing and comparing box-constrained L-BFGS and the fast gradient sign method (FGSM) for generating adversarial examples. The study reveals that box-constrained L-BFGS can generate adversarial examples that are both effective and constrained within a certain input range, ensuring their plausibility and difficulty to detect. Both methods are evaluated against the MNIST digit dataset, with results showcasing their effectiveness in creating adversarial examples that can mislead neural network classifiers. This research contributes to ongoing work in adversarial machine learning, striving to bolster defense mechanisms against adversarial attacks.

1. INTRODUCTION

Adversarial machine learning has emerged as a critical area of research, with a focus on understanding the vulnerabilities of deep learning models and developing robust defenses against adversarial attacks. The inherent susceptibility of neural networks to adversarial examples presents a significant challenge in deploying these models in real-world applications, where security and reliability are of paramount importance. In this paper, we extend on the work of Szegedy et al. [1] and Carlini et al. [2]. We implement box-constrained L-BFGS and fast gradient sign method (FGSM), comparing their ability to generate adversarial examples.

The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm is a widely used quasi-Newton optimization method that has been successfully applied to generate adversarial examples. However, the existing literature has primarily focused on the application of the L-BFGS algorithm in the context of unbounded optimization problems. In this work, we present an implementation of the box-constrained L-BFGS algorithm, which extends its applicability to problems with bounded constraints, such as adversarial examples. This extension allows us to generate adversarial examples that are both effective and constrained within the desired range of input values, ensuring that they remain plausible and difficult to detect.

To evaluate our approach, we utilize the MNIST digit dataset and compare the effectiveness of the box-constrained L-BFGS algorithm against the Fast Gradient Sign Method (FGSM), a popular method for generating adversarial examples. Through qualitative and quantitative visualizations, we assess the performance of both methods in terms of their ability to produce adversarial examples that successfully fool the neural network classifiers.

Our work builds upon the foundations laid by Carlini et al. [2], which explores the properties of adversarial examples and their implications for the robustness of deep learning models. Furthermore, we draw inspiration from the methodology presented in Szegedy et al. [3], which leverages the L-BFGS algorithm for generating adversarial examples.

In the following sections, we describe the methodology employed to implement the box-constrained L-BFGS algorithm and adapt it for use with the cross-entropy loss function. We then present the results of our experiments, comparing the performance of the box-constrained L-BFGS algorithm and the FGSM in generating adversarial examples for the MNIST digit dataset. Through our analysis, we aim to demonstrate the effectiveness of our proposed approach and contribute to the ongoing efforts in developing robust defenses against adversarial attacks in machine learning.

2. BACKGROUND

In this section, we provide a comprehensive background on the two methods for generating adversarial examples: the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm and the Fast Gradient Sign Method (FGSM). We discuss the mathematical formulations and key properties of these methods and cite relevant literature to provide a foundation for our exploration.

A. L-BFGS

The L-BFGS algorithm is an optimization technique that has been widely used in various machine learning applications, including generating adversarial examples that successfully fool neural networks. The method was first introduced by Liu and Nocedal [4] as an adaptation of the BFGS algorithm that is suitable for

large-scale optimization problems. The L-BFGS algorithm has gained popularity due to its effectiveness and efficiency, particularly in the context of non-linear optimization problems.

In the L-BFGS formulation for adversarial examples, the optimization problem is defined as follows:

$$\begin{aligned} &\text{minimize } \|x - x'\|_2^2 \\ &\text{subject to } C(x') = l, \\ &\qquad x' \in [0,1]^n. \end{aligned}$$

The objective function $\|x - x'\|_2$ aims to minimize the Euclidean distance between the original input x and a perturbed input x', such that the classifier $C(x')$ assigns the perturbed input to a target class l. The box constraint $x' \in [0,1]^n$ ensures that the perturbed input remains within the valid range of pixel values, preserving the plausibility of the adversarial example.

In practice, the following optimization problem is easier to solve by using a line search to find an optimal constant $c > 0$:

$$\begin{aligned} &\text{minimize } c \cdot \|x - x'\|_2^2 + \text{loss}_{F,l}(x') \\ &\text{subject to } x' \in [0,1]^n \end{aligned}$$

Here, $\text{loss}_{F,l}(x')$ denotes the loss function of the classifier, which is commonly chosen to be the cross-entropy loss. The L-BFGS algorithm has been successfully applied to generate adversarial examples in several works, such as Szegedy et al. [3] and Goodfellow et al. [4].

B. FGSM

The Fast Gradient Sign Method (FGSM) is another popular technique for generating adversarial examples, which was introduced by Goodfellow et al. [4] as an efficient and effective method to approximate the solution to the adversarial optimization problem. The key idea behind FGSM is to perform gradient ascent on the loss function with respect to the input while using the sign of the gradient instead of the gradient itself. This results in a single-step perturbation of the input that maximizes the classifier's loss.

The FGSM formulation is given by:

$$\vec{x}_{\text{FGSM}} = \vec{x} + \epsilon \text{sgn}(\nabla_{\vec{x}} L(f_\theta(\vec{x}), \vec{y}_{\text{true}})).$$

In this equation, $\vec{x}_{\text{FGSM}}$ denotes the adversarial example, $\vec{x}$ is the original input, ϵ is a small constant controlling the magnitude of the perturbation, $L(f_\theta(\vec{x}), \vec{y}$

This is similar to gradient ascent of the loss with respect to the input, except we only take a single step and use the sign of the gradient instead of the gradient itself.

3. METHODOLOGY

A. L-BFGS

```

def LBFGS(predict, x, y, num_classes=10,
                    binary_search_steps=20,
                    max_iterations=100,
                    initial_const=1e-3,
                    clip_min=0, clip_max=1):

    def compute_distance(x, y):
        diff = (x - y) ** 2
        return diff.view(-1).sum()

    def loss_function(adv_x_np, predict, x,
        target, const):
        adv_x = torch.from_numpy(adv_x_np.
            reshape(x.shape))
            .float().to(x.device).requires_grad_()
        output = predict(adv_x)
        l2_loss = torch.sum((x - adv_x) ** 2)
        cross_entropy_loss =
            F.cross_entropy(output, target)
        scaled_loss = torch.sum(const *
            cross_entropy_loss)
        total_loss = scaled_loss + l2_loss

        total_loss.backward()
        gradient = adv_x.grad.data.cpu().numpy()
            .flatten().astype(float)
        total_loss =
            total_loss.data.cpu().numpy()
            .flatten().astype(float)
        return total_loss, gradient

    x = x.detach().clone()
    y = y.detach().clone()

    c_lower_bound = x.new_zeros(1)
    c_upper_bound = x.new_ones(1) * 1e10
    loss_coeffs = x.new_ones(1) * initial_const
    best_l2_distance = 1e10
    best_label = -1
    best_adversarial = x.clone()
    min_clip = clip_min * np.ones(x.shape[:]).
        astype(float)
    max_clip = clip_max * np.ones(x.shape[:]).
        astype(float)
    clip_bounds = list(zip(min_clip.flatten(),
        max_clip.flatten()))

    for step in range(binary_search_steps):
        initial_guess = x.clone().cpu().numpy()
            .flatten().astype(float)
        adv_x, _, _ =
            fmin_l_bfgs_b(loss_function,
                          initial_guess,
                          args=(predict,
                          x.clone(), y,
                              loss_coeffs),
                          bounds=clip_bounds,
                          maxiter=
                              max_iterations,
                          iprint=-1)

        adv_x = torch.from_numpy(adv_x.reshape(x
            .shape)).float().to(x.device)
        l2_distance = compute_distance(x, adv_x)
        output = predict(adv_x)
        _, output_label = torch.max(output, 1)

        if (l2_distance < best_l2_distance and
            output_label.item() == y.item()):
            best_l2_distance = l2_distance
            best_label = output_label
            best_adversarial = adv_x

        if output_label.item() == y.item():
            c_upper_bound = min(c_upper_bound,
                loss_coeffs)
            if c_upper_bound < 1e10:
                loss_coeffs = (c_lower_bound +
                    c_upper_bound) / 2
        else:
            c_lower_bound = max(c_lower_bound,
                loss_coeffs)
            if c_upper_bound < 1e10:
                loss_coeffs = (c_lower_bound +
                    c_upper_bound) / 2
            else:
                loss_coeffs *= 10

```

```
return best_adversarial
```

Listing 1. LBFGS Algorithm

In the implementation of the L-BFGS algorithm above, we define the following steps:

1. Define a function `compute_distance` that calculates the squared L2 distance between two tensors `x` and `y`.
2. Define a custom loss function `loss_function`, which takes the following inputs: the adversarial example candidate `adv_x_np`, the model's prediction function `predict`, the original input `x`, the true target class `y`, and the constant `const`. Inside this function, we calculate both the L2 loss between the original input and the adversarial example candidate, and the cross-entropy loss between the model's predictions and the true target class. The final total loss is the sum of the scaled cross-entropy loss and the L2 loss.
3. Initialize variables such as the binary search bounds `c_lower_bound` and `c_upper_bound`, the loss coefficients `loss_coeffs`, and the best adversarial example found `best_adversarial`.
4. Iterate through the binary search steps. In each step:
   (a) Set an initial guess for the adversarial example as the original input `x`.
   (b) Use the L-BFGS optimization algorithm `fmin_l_bfgs_b` from SciPy, providing the custom loss function `loss_function`, the initial guess, and other required parameters. This function returns an optimized adversarial example `adv_x`.
   (c) Compute the L2 distance and the model's predictions on the adversarial example `adv_x`.
   (d) If the L2 distance is smaller than the best L2 distance found so far and the model's prediction is incorrect, update the best L2 distance, label, and adversarial example.
   (e) Update the loss coefficients `loss_coeffs` based on whether the model's prediction is correct or incorrect, using the binary search method.
5. Return the best adversarial example found during the search.

B. FGSM

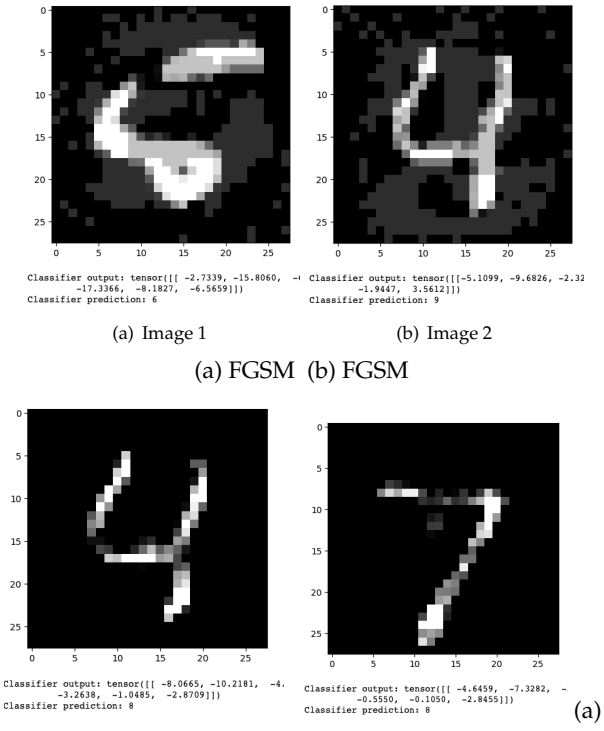
```
def FGSM(x, labels, net, eps):
    '''
    Given an input image X and its corresponding
        labels
    LABELS, as well as a classifier NET, returns
        X
    perturbed by EPS using the fast gradient
        sign method.
    '''
    net.zero_grad()    # Zero out any gradients
        from before
    x.requires_grad=True    # Keep track of
        gradients
    out = net(x)    # Output of classifier
    criterion = nn.CrossEntropyLoss()
    loss = criterion(out, labels)   #
        Classifier's loss
    loss.backward()
    grads = x.grad.data    # Gradient of loss
        w/r/t input

    x_fsgm = x + eps * torch.sign(grads)
    return torch.clamp(x_fsgm, min=-1, max=1)#
        TODO: Your code here!
```

Listing 2. FGSM Algorithm

The Fast Gradient Sign Method (FGSM) implementation in the given code snippet follows a straightforward approach for generating adversarial examples. It computes the gradient of the loss function with respect to the input image, and then perturbs the input image by adding the sign of the gradient multiplied by a small constant epsilon. This approach effectively creates adversarial examples that can mislead the classifier while maintaining a minimal perturbation magnitude.

4. RESULTS



(a) Image 1 (b) Image 2

(a) FGSM (b) FGSM

(c) Image 3 (d) Image 4

(a)

L-BFGS (b) L-BFGS

Fig. 1. Examples comparing both types

A. L-BFGS

We see that the L-BFGS method produced a more convincing adversarial example than FGSM. This is because we continue to optimize the L-BFGS objective, which more thoroughly arrives at the adversarial example than FGSM. In FGSM, after all, we make one alteration to the image whose alteration is parameterized by ϵ. With L-BFGS, on the other hand, we descend for various iterations and arrive at an example that appears more convincing, as we see above.

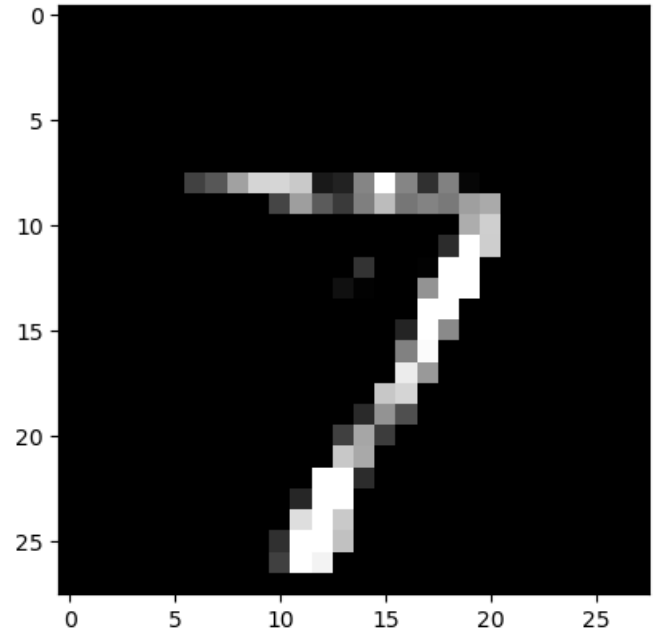


Fig. 2. A successful attack in which the predicted output was 9, not 7, using the L-BFGS method.

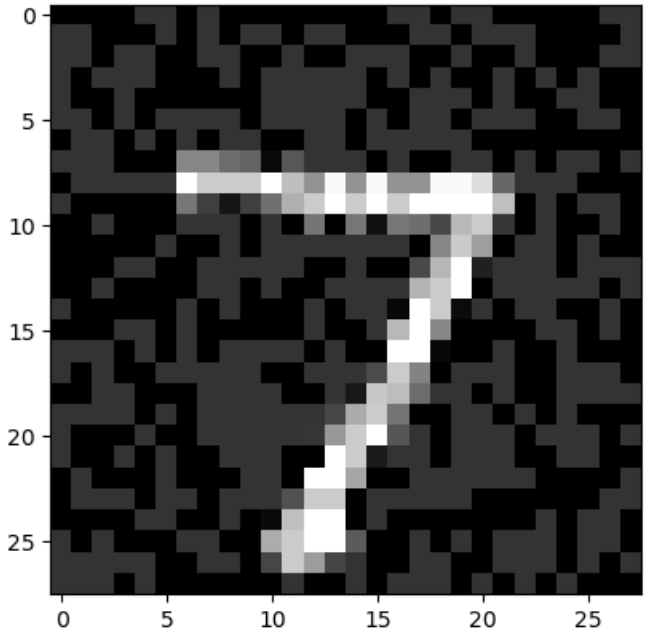


Fig. 3. A successful attack in which the predicted output was 3, not 7, using the FGSM method.

5. DISCUSSION

Qualitatively, FGSM often produces adversarial examples that look spotty and with sharp changes in color due to the nature of the optimization. This can result in images that appear more obviously perturbed to the human eye. Meanwhile, L-BFGS often generates adversarial examples that appear cleaner and more deceptful, as the perturbations are often more subtle and harder to detect by the human eye.

A. Future Work

To provide a better intuition behind how the convex outer bound provably defends against adversarial attacks, we would like to have included a visualization as follows: the user could drag their cursor over the various norms within a defined norm ball that we've seen in the BFGS formulation, as well as the corresponding output in the convex outer bound. This will be worked on in the months to come.

REFERENCES

1. E. Wong and Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," in *Proceedings of the 35th International Conference on Machine Learning*, vol. 80 of *Proceedings of Machine Learning Research* J. Dy and A. Krause, eds. (PMLR, 2018), pp. 5286–5295.
2. N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," CoRR **abs/1608.04644** (2016).
3. C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," arXiv preprint arXiv:1312.6199 (2014).
4. I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," Int. Conf. on Learn. Represent. (2015).