Generating Adversarial Examples for Neural Networks

Mustafa Shaikh

Abstract

In this paper, we explore methods of generating adversarial examples for a neural network trained to classify MNIST data. We implement an epsilon-constrained, cross entropy maximizing algorithm that generates adversarial images to cause the neural network to incorrectly classify MNIST data. We find that our method outperforms the becnchmark, Fast Gradient Sign Method, for several values of the parameter epsilon that governs how different the modified image can be from the original.

1 Introduction

It has been seen that many classifiers, including neural networks, are highly susceptible to adversarial examples – small perturbations of the input that cause a classifier to incorrectly classify the data, but can often be imperceptible to humans (Szegedy et al., 2014). For example, adding a small patch to an image of a stop sign might cause an object detector in an autonomous vehicle to classify it as a yield sign, which could lead to an accident.

2 Formulation and Approach

2.1 Epsilon-constrained, cross-entropy maximization

We now present our approach and the mathematical formulation of the problem. The task is to learn a perturbation vector $r \in \mathbb{R}^{784}$ that, when added to the input image x, causes a reference neural network already trained to classify MNIST data, to misclassify the perturbed input. Inspired by the work by (Szegedy et al., 2014), we aim to directly find the optimum perturbation that maximizes the loss between the true and predicted label. Due to reasons explained below, we reformulate the approach in the cited work to adapt it to our problem.

Our approach works by maximizing the cross entropy loss between the predicted class label for the modified training point, and the true class label, with respect to the perturbation vector. In order to keep modified inputs close to the originals, we impose constraints on the perturbation, and of course constrain the pixel values of the modified grayscale image to fall in the closed interval $[0, 1]$. This leads to a constrained optimization problem that will provide the perturbation that causes the maximum loss between the true and predicted class labels while adhering to problem constraints. It is given by:

$$\max_r -(y \log(\hat{f}(x+r))+ \tag{1}$$
$$+(1-y)\log(1-\hat{f}(x+r))) \tag{2}$$
$$\text{subject to} \tag{3}$$
$$||r|| < \epsilon \tag{4}$$
$$x + r \in [0,1] \tag{5}$$
$$\tag{6}$$

where $\hat{f}(x+r)$ is the one-hot encoded prediction from the pretrained neural network when the input x perturbed by vector r is passed forward through the network. By applying the softmax function to this output vector, the prediction represents the probability that the given data point belongs to class y, i.e. $P(Y = y|X)$. It is given by:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{7}$$

There are some key differences between our approach and (Szegedy et al., 2014), which required us to reformulate the problem. Whereas (Szegedy et al., 2014) seek to force the model to misclassify to a provided label, our aim is simply to misclassify. Furthermore, the authors do not consider a constraint that the modified image must be within epsilon of the original image; instead, they encode this into the objective function.

Due to computational reasons and the highly non-convex nature of the neural network loss, we actually use a slightly simpler version of our approach outlined earlier, which does not guarantee the optimum but is still close and is computationally tractable. We remove the constraints and solve the unconstrained optimization problem using Scipy's Sequential Least Squares solver, and then project the solution into the constraint space. Noting that the constraints are essentially box constraints, this amounts to clipping the values that violate the constraints to the maximum allowed value. For example, if $x + r > 1$ for a given pixel, we clip this to 1. If a given element of r is greater than epsilon, we clip that value to epsilon. Finally, we note that non-convex optimization problems are sensitive to initialization, and we found that initializing with a 0-mean, unit variance Gaussian provided better results than with zeros. This is outlined in Algorithm 1 below.

```
Algorithm 1: $\epsilon$-constrained optimization-based adversarial image generation
Input: $\epsilon$, neural network parameters - W,b,h,z, of layers $\eta$
Output: Modified image
Data: Test set (x,y) pairs - original image x, true class label y
1 Initialize perturbation vector r with 0-mean, unit variance Gaussian;
2 Solve unconstrained optimization problem (remove constraints) ;
3 Project r into epsilon constraint region by clipping to epsilon any values larger than epsilon;
4 Project r into pixel box constraint region by clipping to 0 and 1 any values that lead to $x + r < 0$ and $x + r > 1$ respectively;
5 Add resulting perturbation vector r to input image x to give x+r;
```

2.2 Benchmark - Fast Gradient Sign Method

A popular existing approach, the Fast Gradient Sign Method (FGSM) (Goodfellow et al., 2015), works by taking steps in the direction of the positive gradient of the loss with respect to the *input*, rather than the weights, and is scaled by epsilon in order to satisfy the constraint that each pixel of the modified image must be within epsilon of the original image. We compare our approach with the FGSM method. The update equation for this method is given by:

$$\tilde{x} = x + \epsilon * sign(\nabla_x L(f, x, y)) \qquad (8)$$

where the sign of the gradient is used to ensure a small step is taken in the direction of maximum ascent. We perform two experiments with FGSM; the first using a single gradient ascent update to the input, and the second with 3 gradient ascent steps.

3 Results

First, we show the effect of the perturbation strategy on the actual input image for 2 samples images drawn from the test data. We see that the images generated by our approach are slightly noisier than those generated by FGSM in the region of the digit itself, but are less noisy in the background. This suggests that the algorithm is able to identify that the most important parts of the image to distort are the regions where the digits lie in rather than the background, in order to induce maximum uncertainty in prediction by the neural network. This is an intuitive result and provides confidence that our algorithm is modifying the input in a meaningful manner.

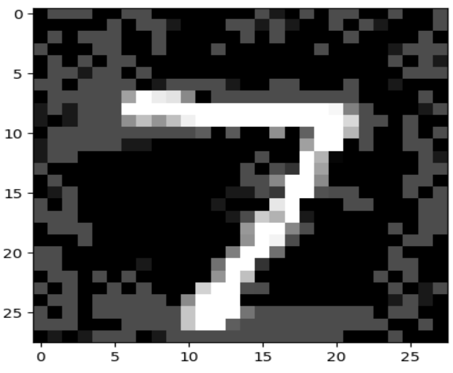

Figure 1: Digit 7 from test set after 3-step FGSM modification

From the graph below, we see that our method

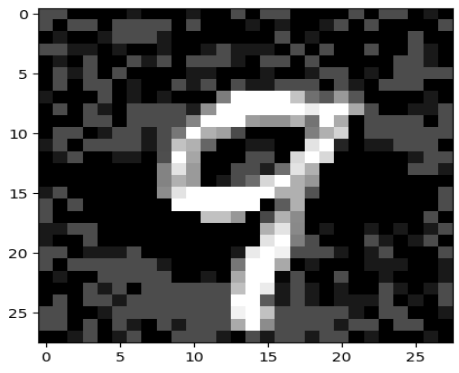

Figure 2: Digit 9 from test set after 3-step FGSM modification

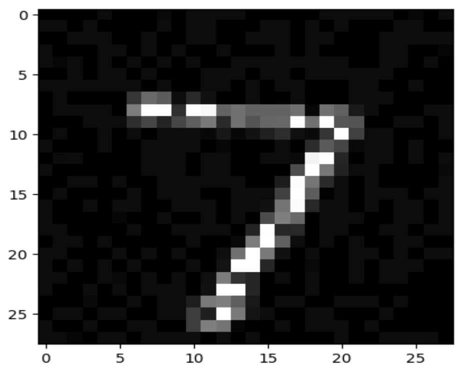

Figure 3: Digit 7 after cross-entropy maximization based approach

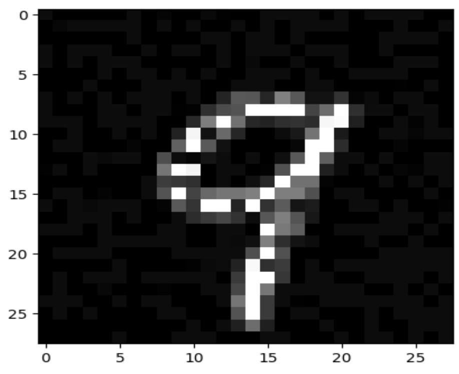

Figure 4: Digit 9 after cross-entropy maximization based approach

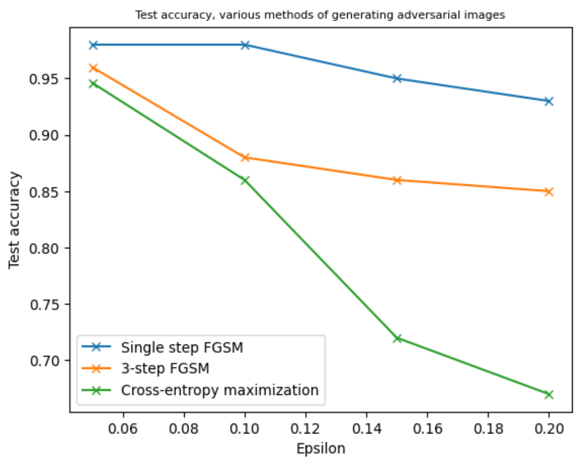

Figure 5: Test accuracy (%) on perturbed inputs, FGSM vs. cross-entropy maximization

clearly outperforms FGSM with both single step and 3-step gradient ascent, across several values of ϵ. In fact, our method steadily decreases test accuracy down to below 70% for $\epsilon = 0.2$, which is significantly lower than FGSM, which does not decrease as much as ϵ increases. It is to be noted however, that our approach incurs significantly higher computational cost. Whereas 3-step FGSM took 30s to run for 200 test data points when evaluating accuracy, our method took over 1.5 hours, which is orders of magnitude slower. This is because an entire optimization problem is solved for each data point, as opposed to a single gradient step for FGSM. It is interesting to note that even solving the unconstrained problem and then projecting the solution into the constraint space can outperform FGSM, even though there is no guarantee of optimality with this approach, which can be considered an approximation of the true optimum at best.

3.1 Next Steps

Our method outperforms the benchmark across a variety of values for ϵ. In future work, due to the high computational cost of this approach, we would explore more efficient approaches that do not require optimization for each data point, such as training another neural network to predict what kinds of perturbations lead to misclassification. Essentially, a dataset would be generated using FGSM or our method, and a neural network trained on this data would be trained to output the optimal perturbation vector when a given input image is passed through the network.

References

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks.

```python
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.optimize import Bounds

def BCE(y_true, y_pred):
     y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)
     return - (y_true @ np.log(y_pred) + (1 - y_true) @ np.log(1
- y_pred))

def oneHotEncode(y):
     tmp = np.zeros((10,))
     tmp[y] = 1
     return tmp

def fwd_attack(x,W1, W2, W3, W4,b1, b2, b3, b4):

        z1 = np.matmul(x,W1)+b1
        h1 = relu(z1)
        z2 = np.matmul(h1,W2)+b2
        h2 = relu(z2)
        z3 = np.matmul(h2,W3)+b3
        h3 = relu(z3)
        z4 = np.matmul(h3,W4)+b4
        p = softmax(z4)
        return p

def gradient(self,x,y):
        '''
        This method finds the gradient of the cross-entropy loss
        of an image-label pair (x,y) w.r.t. to the image x.

        Input
            x: the input image vector in ndarray format
            y: the true label of x

        Output
            a vector in ndarray format representing
            the gradient of the cross-entropy loss of (x,y)
            w.r.t. the image x.
        '''
        p = self.forward(x)
        dLdz4 = p - y
        dz4dh3 = self.W4.T
        dh3dz3_tmp = np.zeros((self.h3.size))
        dh3dz3_tmp[self.h3 > 0] = 1
        dh3dz3 = np.diag(dh3dz3_tmp)
        dz3dh2 = self.W3.T

        dh2dz2_tmp = np.zeros((self.h2.size))
        dh2dz2_tmp[self.h2 > 0] = 1
        dh2dz2 = np.diag(dh2dz2_tmp)
        dz2dh1 = self.W2.T

        dh1dz1_tmp = np.zeros((self.h1.size))
        dh1dz1_tmp[self.h1 > 0] = 1
        dh1dz1 = np.diag(dh1dz1_tmp)
        dz1dh0 = self.W1.T # h0 is x

        return dLdz4 @ dz4dh3 @ dh3dz3 @ dz3dh2 @ dh2dz2 @
dz2dh1 @ dh1dz1 @ dz1dh0


def attackFGSM(self,x,y,num_steps=1):
        '''
        This method generates the adversarial example of an
        image-label pair (x,y).

        Input
            x: an image vector in ndarray format, representing
               the image to be corrupted.
            y: the true label of the image x.

        Output
            a vector in ndarray format, representing
            the adversarial example created from image x.
        '''

        # Fast gradient sign method
        x_til = np.copy(x)
        for _ in range(num_steps):
            x_til += self.eps * np.sign(self.gradient(x_til,y))

        return np.clip(x_til, 0, 1)


@staticmethod
    def perturbationLoss(r,x,y,W1, W2, W3, W4,b1, b2, b3, b4):
        '''
        This method defines the loss function to be optimized to 
find the optimal
        perturbation for input images
        Input: First argument must be the variable of 
optimization i.e. r (perturbation)
            r: perturbation array. size = (784,)
            x: input image. size = (784,)
            y: true class label. size = (1,)
        '''
        return -1 * BCE(y, fwd_attack(x+r,W1, W2, W3, W4,b1, b2, 
b3, b4))

    def myAttack(self,x,y):
        # cons = (
                # {'type': 'ineq', 'fun': lambda r: self.eps - 
np.linalg.norm(r,np.inf)}, # inf norm < eps
                # {'type': 'ineq', 'fun': lambda r: 
np.min(x+r)}, # lb constraint: x+r > 0
                # {'type': 'ineq', 'fun': lambda r: np.min(1-x-
r)}, # ub constraint: x+r<1 i.e. 1-x-r > 0
                # )
        y = oneHotEncode(y)
        r = minimize(self.perturbationLoss, 
x0=np.random.normal(size=(784,)),
                     args=(x,y,self.W1, self.W2, self.W3, 
self.W4,self.b1, self.b2, self.b3, self.b4),
                     method='SLSQP', jac='2-point', 
options={'disp': False}
                    #    , constraints=cons
                       )
        # project the resulting perturbation r onto the 
constraint region;
        # epsilon inf norm constraint and [0,1] pixel value 
constraint
        pert = r.x
        pert[pert > self.eps] = self.eps
        pert = np.where(pert+x >= 1, 1,pert)
        pert = np.where(pert+x <= 0, 0,pert)

        return x + pert

# visualize adversarial image
x,y = X_test[12], Y_test[12]
x_til = clf.myAttack(x,y)
print ("This is an image of Number", y)
pixels = x_til.reshape((28,28))
```

```python
plt.imshow(pixels,cmap="gray")
```

```python
nTest = 100
# list_acc_opt = []
for eps in [0.1,0.15,0.2]:
    print(eps)
    clf.set_attack_budget(eps)
    Y_pred_opt = np.zeros(nTest)
    for i in range(nTest):
        if i%10==0:
            print(i)
        x, y = X_test[i], Y_test[i]
        x_til = clf.myAttack(x,y)
        Y_pred_opt[i] = clf.predict(x_til)
    acc_opt = np.sum(Y_pred_opt == Y_test[:nTest])*1.0/nTest
    list_acc_opt.append(acc_opt)
```

```python
list_acc_fgsm1 = []
for eps in [0.05,0.1,0.15,0.2]:
    print(eps)
    clf.set_attack_budget(eps)
    Y_pred_FGSM = np.zeros(nTest)
    for i in range(nTest):
        x, y = X_test[i], Y_test[i]
        x_til = clf.attackFGSM(x,y,1)
        Y_pred_FGSM[i] = clf.predict(x_til)
    acc_fgsm = np.sum(Y_pred_FGSM == Y_test[:nTest])*1.0/nTest
    list_acc_fgsm1.append(acc_fgsm)
```

```python
plt.plot([0.05,0.1,0.15,0.2], list_acc_fgsm1, marker='x', 
label="Single step FGSM")
plt.plot([0.05,0.1,0.15,0.2], list_acc_fgsm, 
marker='x',label="3-step FGSM")
plt.plot([0.05,0.1,0.15,0.2], list_acc_opt, 
marker='x',label="Cross-entropy maximization")
plt.title('Test accuracy, various methods of generating 
adversarial images',fontsize=8)
plt.ylabel('Test accuracy')
plt.xlabel('Epsilon')
plt.legend()
plt.show()
```