PURDUE UNIVERSITY
INTRODUCTION TO DEEP LEARNING

Project 4: Adversarial Machine Learning

1 INTRODUCTION

Machine learning models have recently been found to be vulnerable to *adversarial attacks*, which are data samples that, although appear unaltered to a human, cause high-confidence misclassifications on well-trained deep learning classifiers. For example, as shown in Figure 1.1, an image that would be correctly classified by a trained network is classified as something completely different with the addition of a visually imperceptible perturbation. As a result, adversarial attacks pose a serious threat to machine learning models. Although several algorithms have been proposed to defend deep learning models from these malicious attacks (as we will explore in this project), finding a solution to completely mitigate both the causes and effects of adversarial attacks remains an open question in the machine learning research community.

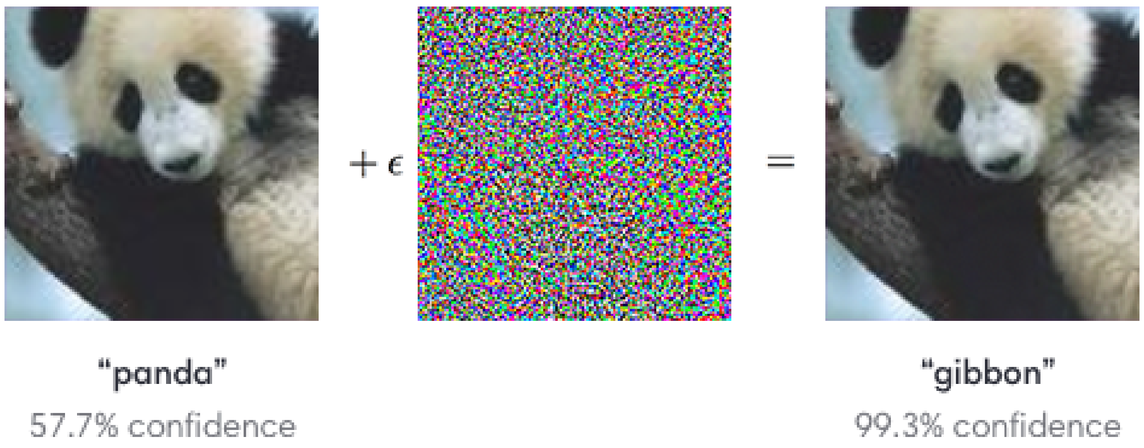


Figure 1.1: An adversarially perturbed image with $\epsilon = 0.007$. The initial image is correctly classified with 57.7% confidence, the adversarial noise is incorrectly classified as a nematode with 8.2% confidence, and the final adversarial image is incorrectly classified as a gibbon with 99.3% confidence.

Throughout this project, we will explore four of the most common adversarial attack algorithms as well as four common defenses. We will implement each of the discussed attacks on a trained artificial neural network using the Cleverhans library, which is used as a benchmark for imposing attacks and observing their effects. We will then implement one of the four discussed defenses for each attack and observe its effect on the classification accuracy. For more information on Cleverhans, see its documentation (`https://cleverhans.readthedocs.io/en/latest/source/attacks.html`) and its open source code (`https://github.com/tensorflow/cleverhans/tree/master/cleverhans/attacks`).

1.1 Standard Adversarial Attacks

1.1.1 The Fast Gradient Sign Method

The Fast Gradient Sign Method (FGSM) is one of the most computationally efficient adversarial attacks to implement. It is implemented by computing the sign of classifier's cost function's gradient and adding a certain error to each element in the data sample consistent with the direction of the sign. Mathematically, this can be represented by the following equation:

$$\widetilde{x} = x + \epsilon sgn(\nabla_x J(w, x, y)). \tag{1.1}$$

In (1.1), x refers to the original input sample and $sgn(\nabla_x J(w, x, y))$ is the sign of the gradient of the cost function $J(w, x, y)$, which is a function of the input, x, the output classification, y, and the classifier weights, w. The parameter ϵ is the l_∞-bounded perturbation chosen by the attacker and given by $\epsilon = \|x - \widetilde{x}\|_\infty$. Note that the perturbation is uniformly added to each dimension in x.

1.1.2 Projected Gradient Descent

Projected Gradient Descent (PGD) is very similar to the FGSM attack; however, unlike the FGSM attack, PGD is iterative. Instead of calculating the gradient of the cost function once and adding a larger bounded error, PGD calculates a new gradient in each iteration and adds a much smaller perturbation to each point in the data sample. Mathematically, PGD can be represented as follows:

$$x^{t+1} = \Pi_{x+s}(x^t + \alpha sgn(\nabla_x J(w, x, y))). \tag{1.2}$$

In (1.2), x^{t+1} refers to the data sample at the current iteration, x^t refers to the data sample at the previous iteration, S is the set of all samples in the dataset, and α is a small perturbation constant. Similiar to the FGM attack, $sgn(\nabla_x J(w, x, y))$ refers to the gradient of the cost function w.r.t. the data sample.

1.1.3 The Carlini and Wagner Attack

The Carlini and Wagner attack is an iterative attack that adds a perturbation to a data sample under the l_2 distance constraint. Given a target class, t, which is not equal to the true class, i, the Carlini and Wagner Attack aims to find w in the following optimization problem using gradient descent:

$$\text{minimize} \quad \left\| \tfrac{1}{2}(\tanh(w) + 1) - x \right\|_2^2 + c \cdot f(\tfrac{1}{2}\tanh(w) + 1)$$

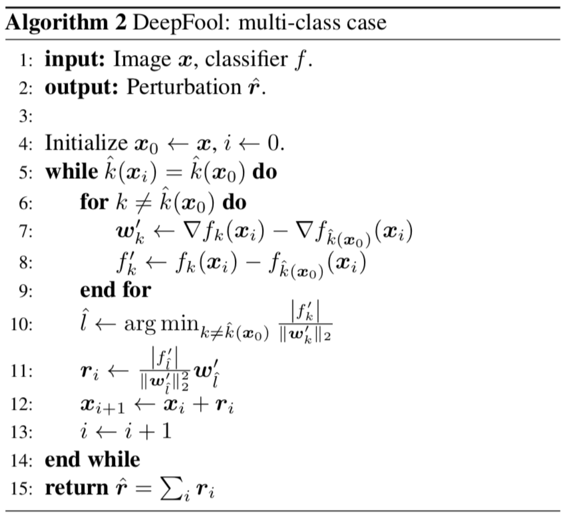
with f defined as (1.3)

$$f(x') = max(max(\{Z(x')_i : i \neq t\} - Z(x')_t, -\kappa)$$

In (1.3), x is the true sample, x' is the adversarial sample, c is a suitably chosen constant (which is empirically determined), κ is a constant that allows the adversarial sample to be misclassified with high confidence, and $Z(\cdot)$ is the output of the classifier before applying the softmax activation (also called logits). This attack is one of the most effective adversarial attacks, but it requires a high computational cost and, as a result, takes much more time to implement in comparison to other adversarial attacks.

1.1.4 DeepFool

DeepFool is an iterative attack in which a very small perturbation is added to a sample and then the sample is classified to determine if it is adversarial. If the sample is adversarial, the loop terminates on that particular sample. Otherwise, another tiny perturbation is added. This loop continues for a given number of maximum iterations or until the sample is misclassified (which ever occurs first). The entire DeepFool algorithm for the multi-class case (taken directly from the paper that introduced it) is shown in Figure 1.2 below.



Algorithm 2 DeepFool: multi-class case

```
input: Image $\boldsymbol{x}$, classifier $f$.
output: Perturbation $\hat{\boldsymbol{r}}$.

Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
while $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ do
    for $k \neq \hat{k}(\boldsymbol{x}_0)$ do
        $\boldsymbol{w}'_k \leftarrow \nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
        $f'_k \leftarrow f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
    end for
    $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x}_0)} \frac{|f'_k|}{\|\boldsymbol{w}'_k\|_2}$
    $\boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{\|\boldsymbol{w}'_{\hat{l}}\|_2^2} \boldsymbol{w}'_{\hat{l}}$
    $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
    $i \leftarrow i + 1$
end while
return $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$
```

Figure 1.2: DeepFool algorithm where k is the true class, $\hat{k}$ is the predicted class, w is the classifier weights, and f is the classifier

1.2 Selected Defenses

1.2.1 Denoising Autoencoders

As we saw in Project 2, Denoising Autoencoders (DAEs) can be extremely effective in removing Gaussian noise from data. However, DAEs can be used in a much broader context than only removing Gaussian noise. In fact, they can be used to undo any corruption process in which unwanted perturbations were introduced into data. As a result, DAEs have been utilized as a pre-processing mechanism to remove adversarial noise from data before sending it to the classifier. To train the DAE, adversarial perturbations are generated on a pre-trained classifier. Then, these adversarial samples are used, with their corresponding uncorrupted inputs as labels, to train the DAE to remove the adversarial perturbations. Typically, the autoencoders are also trained to reconstruct the original training data in order to retain robustness on data in the absence of an attack.

1.2.2 Adversarial Training

Adversarial training is a widely proposed defense that suggests generating adversarial examples using the gradient of the target classifier that will be attacked, and then re-training the classifier with the adversarial samples and their respective corrected labels. This method has proven to be effective in defending classifiers from adversarial attacks, but it is associated with high computational costs (i.e., re-training a large classifier may not always be feasible) and leaves the classifier vulnerable to more potent attacks.

1.2.3 Dimensionality Reduction (e.g. PCA)

Dimensionality reduction can be imposed in a variety of ways but, in the context of developing defensive algorithms to adversarial attacks, Principal Component Analysis (PCA) is one of the most common dimensionality reduction defenses. The PCA algorithm for defending machine learning models from adversarial attacks works as follows:

1. Normalize the training and testing data
2. Calculate the covariance matrix using the training data
3. Calculate the eigenvalues and eigenvectors
4. Select the number of principal components to form feature vectors
5. Train a new classifier using the feature vectors from the preceding step
6. Project the testing data onto the same subspace obtained from the training data and form the appropriate dimensional (same number of dimensions as was chosen to train the classifier) feature vectors
7. When classifying the adversarial data, project the data onto the lower dimensional subspace and use its corresponding classifier for classification

PCA works well in defending certain types of attacks and works less effectively when defending others. However, this method can again be circumvented if an attacker is aware that a different classifier is being used. In such a scenario, the adversary could target their attack to the classifier trained on the principal components.

Note: You can use scikit-learn's (or any ML library's) PCA implementation in lieu of following steps 1 - 7 above.

1.2.4 Adversarial Detection

Traditionally, defenses against adversarial attacks are associated with correctly classifying maliciously altered input samples. However, detecting that an example may be adversarial can be equally important (i.e., if there is a mechanism in place to detect that an example contains adversarial noise, the system can be told not to trust its respective classification). Adversarial detection can be implemented using various techniques, but, in this project, we will attempt to detect them at the input before the samples are propagated for classification. We will do this by training an autoencoder (using the mean squared error loss) on the training set, generating adversarial examples using the training set, predicting the reconstructions of the adversarial examples from the training set using the autoencoder, and setting the smallest obtained reconstruction error as the threshold. Then, our detection method will work as follows: for every sample in the testing set, determine the sample's reconstruction error (using the autoencode). If that error is greater than the threshold, consider the example adversarial. In this project, we will count how many true positive and false positive examples we get. Ideally, we want 0 false positives and 10,000 (since that is the number of samples in the testing set) true positives.

1.3 Overview

The objective of this project is to implement the four aforementioned attacks and defenses on a classifier trained to classify hand-written digits from the MNIST database. We will begin by training a target classifier whose gradient will be used to construct the various attacks. Then, we will observe the effect of different adversarial perturbations on a trained classifier with high accuracy on the original testing set. Lastly, each of the attacks will be accompanied by a defense, and the effect of the defense will be observed in relation to using no defense. You will notice that after implementing each of the four attacks, the perturbed samples will look almost indistinguishable from their original counter parts. This hidden-in-plain-sight attack is responsible for raising the security concerns associated with adversarial deep learning attacks.

For this project, the MNIST data can be loaded using Keras (as in Project 2 and Project 3). You are expected to fill in the Jupyter Notebook provided to you as instructed in each of the following sections. **Submit your final code as a single .ipynb file and as a .pdf file of the notebook. Ensure that the deliverables are clearly shown for each part to earn full credit. Name your submission *lastname_firstname_aml.ipynb*.**

2 EXAMPLE CLEVERHANS CODE

The following listing shows how to implement the L-BFGS on a trained Keras neural network classifier using Cleverhans. This was an attack specifically developed for attacking convolutional neural networks. The syntax of this listing should be adopted when implementing standard attacks in your own code.

Listing 1: Implementing the LBFGS attack using Cleverhans

```
#ECE 595 Machine Learning II
#Project 4: Adversarial ML - Example Code
#Written by: Rajeev Sahay and Aly El Gamal

import keras
from keras.datasets import mnist
from keras.models import load_model
from keras import backend
from cleverhans.utils_keras import KerasModelWrapper
from cleverhans.attacks import LBFGS


#Import dataset and normalize to [0,1]
(_, _), (data_test, labels_test) = mnist.load_data()
data_test = data_test/255.0

#Reshape to 784-dimensional vectors
data_test = data_test.reshape(10000, 784)

#Create labels as one-hot vectors
labels_test = keras.utils.np_utils.to_categorical(labels_test, num_classes=10)

#Import pre-trained classifier
mnist_classifier = load_model('mnist_classifier.h5')

#Get TensorFlow Session to pass into Cleverhans modules
sess = backend.get_session()

#Create wrapper for classifier model so that it can be passed into Cleverhans modules
wrap = KerasModelWrapper(mnist_classifier)



#Implementing the LBFGS attack

#LBFGS Instance on trained mnist_classifier
lbfgs = LBFGS(wrap, sess=sess)

#Attack parameters
lbfgs_params = {'batch_size':150,
                'binary_search_steps': 1,
                'max_iterations': 50,
                'clip_min': 0.,
                'clip_max': 1.}


#Generate adversarial data
lbfgs_attack_data = lbfgs.generate_np(data_test, **lbfgs_params)


#Evaluate accuracy of perturbed data on target classifier
lbfgs_adv_scores = mnist_classifier.evaluate(lbfgs_attack_data, labels_test)
print("LBFGS Adversarial Attack Accuracy: %.2f%%" %(lbfgs_adv_scores[1]*100))
```

3 Part 1: Training a Target Classifier

Begin by training a simple artificial neural network (according to the procedure below) to classify hand written digits from the MNIST database. This trained classifier will serve as the target classifier to attack for the remaining sections

1. Load the MNIST data from Keras
2. Normalize each sample in the training and testing set to lie in $[0, 1]$. Note that the MNIST data lies in the range $[0, 255]$.
3. Reshape both the training and testing data into 60000×784 and 10000×784 matrices, respectively.
4. Convert the training and testing labels into one-hot vectors.
5. Create a Sequential model with the following parameters:
    - First hidden layer: a dense (fully-connected) layer consisting of 100 units with a ReLU activation and an input dimension of 784
    - a batch normalization layer
    - Second hidden layer: a dense (fully-connected) layer consisting of 100 units with a ReLU activation
    - a batch normalization layer
    - Output layer: a dense (fully-connected) layer consisting of 10 units (the predicted classifications) with a softmax activation
6. Compile the model using the categorical cross entropy loss, the Adam optimizer, and using the 'accuracy' metric.
7. Train the model using the MNIST training data, 50 epochs, and a batch size of 256.
8. Show a plot of the model loss versus the epochs during training.
9. Show a plot of the model accuracy versus the epochs during training.
10. Print the accuracy of the trained classifier on the MNIST testing data.

Deliverables for Part 1:

1. Show a plot of the model loss versus the epochs during training.
2. Show a plot of the model accuracy versus the epochs during training.
3. Print the accuracy of the trained classifier on the MNIST testing data.

4 Part 2: The Fast Gradient Method

Use the gradient of the target classifier from Part 1 to create a testing set (of MNIST digits) with adversarial perturbations according to the Fast Gradient Method. Follow the procedure below.

1. Instantiate a FastGradientMethod instance using cleverhans.attacks.FastGradientMethod(wrap, sess=sess)
2. Setup the following attack parameters: eps = 0.25 (the l_∞-bounded perturbation), clip_min = 0.0, and clip_max = 1.0.
3. Generate adversarial perturbations
4. Evaluate the accuracy of the adversarial testing set on the classifier
5. Show ten samples from the original testing set and their corresponding samples after adding the adversarial perturbations
6. Implement the autoencoder based detection by

   - Training an autoencoder using 50 epochs
   - Using the training set to generate 60,000 adversarial examples using the gradient of the classifier from Part 1 and an l_∞ bound of 0.25.
   - Generating reconstructions of the 60,000 adversarial examples using the trained autoencoder (**Hint:** Use autoencoder.predict(adversarial_data).)
   - Calculating the mean squared error between the adversarial inputs and their reconstructions (**Hint:** First use 'error = keras.losses.mean_squared_error(x, y),' which will return a tensor and then use 'error = error.eval(session=sess)' to convert the tensor into a NumPy array. The parameter 'sess' will come from above when it was defined for generating the adversarial attack so there is no need to re-define it.)
   - Finding the minimum error
   - Setting the minimum error to a threshold
   - Getting the reconstructions of the adversarial testing set (again by using model.predict())
   - Calculating the error of the testing set
   - Calculating the printing the number of true positives
   - Calculating and printing the number of false positives.

Deliverables for Part 2:

1. Print the accuracy of the adversarial testing set.
2. Show ten sample images from the testing set before adding the adversarial perturbation.
3. Show the corresponding ten sample images from the testing set after adding the adversarial perturbation.
4. Print the threshold value.
5. Print the number of true positives.
6. Print the number of false positives.

5 Part 3: Projected Gradient Descent

Use the gradient of the target classifier from Part 1 to create a testing set (of MNIST digits) with adversarial perturbations according to the Projected Gradient Descent attack. Follow the procedure below.

1. Instantiate a ProjectedGradientDescent instance using cleverhans.attacks.MadryEtAl(wrap, sess=sess)
2. Setup the following attack parameters: eps = 0.25 (the maximum allowed l_∞-bounded perturbation from the original point), eps_iter = 0.01 (the amount to move in the direction of the gradient on each iteration), nb_iter = 20 (the number of iterations of PGD), clip_min = 0.0, and clip_max = 1.0.
3. Generate adversarial perturbations
4. Evaluate the accuracy of the adversarial testing set on the classifier
5. Show ten samples from the original testing set and their corresponding samples after adding the adversarial perturbations
6. Implement the adversarial training defense (assume that the attacker cannot instantiate an attack further than what was generated in #3) by generating adversarial perturbations (with eps = 0.25 and eps_iter = 0.01) on the training set and then re-training the classifier with this and the original training set and their corresponding corrected labels using 50 epochs. **Ensure not to overwrite the original classifier by first assigning the current classifier to a new name.** For example, if the trained model from Part 1 is saved as the variable 'classifier,' then use the following line of code:

   ```
   adv_trained_clf = classifier
   ```

   [*1*]

   Now train 'adv_trained_clf' on the generated adversarial data.
7. Test the effectiveness of the re-trained network by evaluating the accuracy of the adversarial data generated in #3. Print the accuracy of the classifier, after re-training it, on the perturbed data from #3.

Deliverables for Part 3:

1. Print the accuracy of the adversarial testing set.
2. Show ten sample images from the testing set before adding the adversarial perturbation.
3. Show the corresponding ten sample images from the testing set after adding the adversarial perturbation.
4. Print the accuracy of the adversarial testing set when using the re-trained network.

6 Part 4: The Carlini and Wagner Attack

Use the gradient of the target classifier from Part 1 to create a testing set (of MNIST digits) with adversarial perturbations according to the Carlini and Wagner attack. Follow the procedure below.

1. Instantiate a Carlini and Wagner attack instance using cleverhans.attacks.CarliniWagnerL2(wrap, sess=sess)
2. Setup the following attack parameters: binary_search_steps = 1 (the number of times binary search is performed to find the optimal tradeoff constant between norm of the perturbation and confidence of the classification), y = None (the classification labels which will not be supplied), learning_rate = 1.25 (the gradient descent learning rate), batch_size = 16, initial_const = 10, clip_min = 0.0, and clip_max = 1.0.
3. Generate adversarial perturbations on the testing set
4. Evaluate the accuracy of the adversarial testing set on the classifier
5. Show ten samples from the original testing set and their corresponding samples after adding the adversarial perturbations
6. Implement the dimensionality reduction defense (assume that the attacker cannot instantiate an attack further than what was generated in #3) by training a new model (according to the following specification) on the first 100 principal components of each sample in the original training set. **Note that the original training and testing data has been transformed into its principal components in the given Jupyter Notebook**.
   - First hidden layer: a dense (fully-connected) layer consisting of 100 units with a ReLU activation and an input dimension of 100
   - a batch normalization layer
   - Second hidden layer: a dense (fully-connected) layer consisting of 100 units with a ReLU activation
   - a batch normalization layer
   - Output layer: a dense (fully-connected) layer consisting of 10 units (the predicted classifications) with a softmax activation
   - Compile your model using the categorical crossentropy loss and adam optimizer
   - For training, use a batch size of 256 and 50 epochs
7. Test the effectiveness of the defense by transforming the perturbed dataset from #3 into its principal components and evaluating its accuracy on the new model trained using the principal components of the training data. Print the accuracy of the classifier on the perturbed data from #3.

Deliverables for Part 4:

1. Print the accuracy of the adversarial testing set.
2. Show ten sample images from the testing set before adding the adversarial perturbation.
3. Show the corresponding ten sample images from the testing set after adding the adversarial perturbation.
4. Print the accuracy of the adversarial testing set when using the dimensionality reduction defense.

7 Part 5: DeepFool

Use the gradient of the target classifier from Part 1 to create a testing set (of MNIST digits) with adversarial perturbations using the DeepFool attack. Follow the procedure below.

1. Instantiate a DeepFool instance using cleverhans.attacks.DeepFool(wrap, sess=sess)
2. Setup the following attack parameters: nb_candidate = 10 (the number of classes), max_iter = 50 (the number of iterations, algorithm will terminate when sample is misclassified or when the maximum number of iterations occur, which ever occurs first), clip_min = 0.0, and clip_max = 1.0.
3. Generate adversarial perturbations
4. Evaluate the accuracy of the adversarial testing set on the classifier
5. Show ten samples from the original testing set and their corresponding samples after adding the adversarial perturbations
6. Implement the Denoising Autoencoder Defense (assume that the attacker cannot instantiate an attack further than what was generated in #3) by generating adversarial perturbations (with an l_∞-bounded perturbation of 0.25) on the training set and then training an autoencoder to remove the adversarial perturbation. Use 50 epochs for training the DAE.
7. Test the effectiveness of the DAE by propagating the attacked data from #3 through the DAE before evaluating the accuracy from the target classifier. Print the accuracy of the classifier after pre-processing the perturbed data.
8. Show ten samples of the adversarial data after processing it through the DAE.

Deliverables for Part 5:

1. Print the accuracy of the adversarial testing set.
2. Show ten sample images from the testing set before adding the adversarial perturbation.
3. Show the corresponding ten sample images from the testing set after adding the adversarial perturbation.
4. Print the accuracy of the testing set after it has been forward propagated through the DAE.
5. Show ten samples of adversarial examples after denoising.