Notes for a Workshop on Adversarial AI focused on Evasion

Ricardo Morla
ricardo.morla@fe.up.pt

Faculty of Engineering and INESC TEC, University of Porto, Portugal
March 2019

Abstract. This workshop is aimed at people interested in machine learning and cybersecurity, specifically in the security of machine learning and in the implications of using machine learning in cybersecurity. It does not require any previous experience with machine learning, although experience with designing and programming algorithms and with multivariate calculus can make everyone's life easier.

1 Introduction

Although an increasing number of business models and security tasks are based on machine learning, the overwhelming majority of what we do today with machine learning does not consider a motivated attacker. Known threat models for machine learning include the creation of adversarial samples for inference (known as evasion attacks) and learning (known as poisoning attacks), stealing of previously trained models (by calls to the inference API or sidechannel attacks to the inference software and hardware), and identifying which samples were used in the learning (that may compromise e.g. the privacy of the users that generated those samples). This workshop is a hands-on introduction to the topic of adversarial learning focused on generating adversarial samples for inference. We start by a practical introduction to tensorflow on Docker, then explain the mathematical models of layered neural networks with a 2-layer example for which we derive and implement the feedforward computational graph that performs inference on tensorflow. Then we illustrate how the backpropagation and gradient descent mechanisms can be used to learn the parameters of the neural network from training data and improve the accuracy of the inference on test data. After deriving the mathematical expressions for the gradients we show how to implement the gradient graph on tensorflow and how to run it in order to learn the parameters of the network. Backpropagation can be used not only to learn the parameters but also to learn adversarial samples. We derive the expression for the gradient with regard to the input data and implement a computational graph to generate adversarial samples for inference. Finally, we provide pointers to other adversarial learning approaches and threat models, and how these approaches can be applied in different aspects of cybersecurity such as malware and C2C traffic detection.

Refer to Goodfellow et al. [1] for foundation and technical details on deep learning. Refer to Vorobeychik et al. [13] and to Joseph et al. [6] for details on adversarial learning.

1.1 Time management

This workshop is planned for 3 hours as follows:

- First hour – Setup tensorflow on docker and explore loading data and tensorflow basics, sections 2 and 3
- Second hour – Learn how to do inference and learning on a 2 layer dense neural network, sections 4 and 5
- Third hour – Explore adversarial attacks on inference of the 2 layer network, sections 6 and 7.

2 Setting up Tensorflow under Docker

Make sure you have a docker setup in your machine. Follow this tutorial[1] if you don't. This assumes an Ubuntu 18-04 server box in hardware or VM[2].

2.1 Create a Dockerfile

Create a Dockerfile from tensorflow that includes Keras and CleverHans.

```
$ vim Dockerfile-advai
>>
FROM tensorflow/tensorflow:1.13.1-py3-jupyter
RUN pip install --upgrade keras
RUN pip install --upgrade flask-restful
RUN pip install --upgrade cleverhans
<<
```

2.2 Build a Docker image

Before building make sure you don't have what you want in your system yet.

```
$ docker images
$ docker build --tag tensorflow/tensorflow:advai - < Dockerfile-advai
```

[1] https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04

[2] e.g. https://www.virtualbox.org

2.3 Run a Docker container, copy notebook app token

Before running make sure you don't have a container running that does what you want, or kill and remove it before running again.

```
$ docker ps
$ docker kill advaicontainer ; docker rm advaicontainer
$ docker run -u $(id -u):$(id -g) --name advaicontainer -it --rm
   -v /home/keras/code:/tf/code  -v /home/keras/data:/tf/data
   -p 8888:8888 tensorflow/tensorflow:advai
```

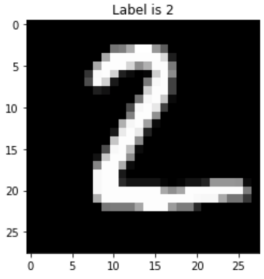
2.4 Open the Jupyter notebook on your browser and plot the number images

Point your browser to the notebook app token URL. The token is output when you run the container at http://localhost:8888/?token=xxxxxxxx.

To check if everything is working and to prepare the datasets you will be using next, download the MNIST train and test data sets from here[3] into your `/home/keras/data` folder. The load the number images and display them as follows.

```
import numpy as np
from matplotlib import pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tf/data/mnist/", one_hot=True)
image = mnist.test.images[0]
image_class = np.argmax(mnist.test.labels[0])
image_2d = image.reshape(28, 28)
plt.title('Label is {label}'.format(label=image_class ))
plt.imshow(image_2d, cmap='gray')
plt.show()
```

Fig. 1. Example of an MNIST image and label.



[3] http://yann.lecun.com/exdb/mnist/

3 Tensorflow basics

3.1 Computational graphs

Simple computational graph example (https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/1_Introduction/basic_operations.py).

```
import tensorflow as tf
# define computational graph
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)
add = tf.add(a, b)
mul = tf.multiply(a, b)
# process data using previously defined computational graph
with tf.Session() as sess:
    add_result = sess.run(add, feed_dict={a: 2, b: 3})
    mul_result = sess.run(mul, feed_dict={a: 2, b: 3})
    print add_result, mul_result
```

3.2 Tensor shapes

Tensor shape examples [4]

- Tensor [4] → shape [] (scalar)
- Tensor [1, 2, 3, 4, 5, 6, 7, 8, 9] → shape [9] (vector)
- Tensor [[1, 2, 3], [4, 5, 6], [7, 8, 9]] → shape [3, 3] (matrix)
- Tensor [[[1, 2, 2], [3, 4, 1]], [[5, 6, 5], [7, 8, 9]]] → shape [2, 2, 3] (just a tensor)

Matrix multiplication computational graph example.

```
x = tf.truncated_normal([1024, 512])
y = tf.truncated_normal([512, 1024])
z = tf.matmul(x,y)
```

3.3 TensorFlow and numpy

Compatible tensor notation. No computational graphs in numpy.

4 Neural networks – Feedforward

4.1 Mathematical model of a layer

A dense neural network can be used to approximate any function F by taking the input data at the lowest layer and producing the output at the highest layer. The

[4] https://www.tensorflow.org/api_docs/python/tf/reshape

output of one layer is the input to the next layer. The neural network has parameters at each layer. More details on the model and mathematical formulation can be found on this online book[5].

At layer l, the output j given input k from the previous layer is: $a_j^l = \sigma(z_j^l)$ with $z_j^l = \sum_k a_k^{l-1} w_{kj}^l + b_j^l$. Here we define $\sigma(x) = 1/(1+e^{-x})$ as the non-linearity applied to the z_j^l linear combination with weights w_{kj}^l and bias b_j^l parameters. Using the vectorized form of $\sigma(z^l)$ as a vector with elements $\sigma(z_j^l)$ we get the vector expression for the layer equation $a^l = \sigma(a^{l-1} w^l + b^l)$.

4.2 A 2-layer network

The following represents the layers of a 2-layer network. $a(x; w^1, b^1, w^2, b^2) = a^2$ with $a^2 = \sigma(a^1 w^2 + b^2)$, $a^1 = \sigma(a^0 w^1 + b^1)$, and $a^0 = x$.

The input to the network is a^0 with e.g. shape [1, 784] for a 28 x 28 pixel MNIST image. The output of the network is a^2, e.g. with shape [1, 10] for a one-out-of-10 classes that the network outputs for each a^0 input. The shape of the parameters is related to the structure of the network. For example the number of intermediate neurons (e.g. 128) defines the shapes of a^1 and b^1 (both shape [1, 128]) and of w^1 (shape [784, 128]). The number of outputs of the network defines the shapes of a^2 and b^2 (both shape [1, 10]) and of w^2 (shape [128, 10]).

4.3 Feedforward on TensorFlow

Define placeholders for input (x) and output (y) data. Input data placeholders are defined for a single [784,1] image tensor and output data placeholders are defined for a single [10,1] image class tensor. Additionally, set (random) values for all the weights. Because the weights are random the accuracy of the neural network is likely not great. Define σ non-linear activation and the feedforward propagation function that returns the value of a for each input. More details here[6].

First define the variables and specify the computational graph.

```
# variables
x = tf.placeholder(tf.float32, [1, 784])
y = tf.placeholder(tf.float32, [1, 10])
w_1 = tf.Variable(tf.truncated_normal([784, 128]))
b_1 = tf.Variable(tf.truncated_normal([1, 128]))
w_2 = tf.Variable(tf.truncated_normal([128, 10]))
b_2 = tf.Variable(tf.truncated_normal([1, 10]))
# computational graph
def sigma(x):
   return tf.div(tf.constant(1.0), tf.add(tf.constant(1.0),
        tf.exp(tf.negative(x))))
a_0 = x
```

[5] http://neuralnetworksanddeeplearning.com
[6] http://blog.aloni.org/posts/backprop-with-tensorflow/

```
z_1 = tf.add(tf.matmul(a_0, w_1), b_1)
a_1 = sigma(z_1)
z_2 = tf.add(tf.matmul(a_1, w_2), b_2)
a_2 = sigma(z_2)
# a: output of the feedforward neural network with input data x
a = a_2
# accuracy
acct_mat = tf.equal(tf.argmax(a, 1), tf.argmax(y, 1))
acct_res = tf.reduce_sum(tf.cast(acct_mat, tf.float32))
```

Finally load the MNIST data set and run the graph.

```
# load images
mnist = input_data.read_data_sets("/tf/data/mnist/", one_hot=True)
one_image, one_class = mnist.train.next_batch(1)
# run session
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    [acct_res_output, a_output] = sess.run([acct_res, a],
            feed_dict={ x : one_image, y : one_class })
    print(acct_res_output, a_output)
```

Challenge: redo for multiple images.

5 Neural networks – Backpropagation

We want to use existing X and Y data to have better predictions of $F(X)$ for new values of X. For this we can assume the input and output data X, Y are fixed and use the set of parameters $p = \{w^1, b^1, ..\}$ as variables of our neural network a. The goal is to understand how we can adjust the parameter values – starting from random values – in order to reduce the error of $a(X)$ against Y, i.e. the difference between the output data Y and the output of function $a(X)$ at the corresponding input point X. The method that is typically used is called gradient descent, and to compute the gradient we use backpropagation and the chain rule. This part is based on chapter two of this online book[7].

5.1 Chain rule

The chain rule lets us compute the gradient without knowing the exact form of the intermediate function, just the derivative.

- With $f, g : \mathbb{R} \to \mathbb{R}$ we get $(f(g(x))' = f'(g(x)) \cdot g'(x)$.
- With functions with more input and output variables, then the chain rule becomes $\frac{\partial f_i}{\partial x_j}(x) = \sum_k \frac{\partial f_i}{\partial g_k}(g(x)) \cdot \frac{\partial g_k}{\partial x_i}(x)$

Note that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

[7] http://neuralnetworksanddeeplearning.com

5.2 Backpropagation

Consider the error function $e(p; x, y) = \frac{1}{2}(y - a(p; x))^2$ with neural network parameters p as variables and x, y input and output data as parameters. We need to understand how to change each parameter in p to reduce the error. For that we get the derivative of e with regard to the top layer z^L, and drill down through the layers until we get the expressions for how e varies with each parameter. In particular we look at how the error varies with z^l at layer l as $\delta^l_j = \frac{\partial e}{\partial z^l_j}$.

- For the top layer L we get $\delta^L_j = \frac{\partial e}{\partial z^L_j} = \frac{\partial e}{\partial a^L_j} \cdot \frac{\partial a^L_j}{\partial z^L_j} = (a^L_j - y_j) \cdot \sigma'(z^L)$. In vector form this becomes $\delta^L = \nabla_a e \odot \sigma'(z^L) = (a^L - y) \odot \sigma'(z^L)$. Notice the Hadamar product $f \odot g$, which is the element-by-element multiplication of two tensors.
- For the other layers we can write how the error in that layer δ^l varies with the variation of the error from the upper layer δ^{l+1}. This means $\delta^l_j = \frac{\partial e}{\partial z^l_j} = \sum_k \frac{\partial e}{\partial z^{l+1}_k} \cdot \frac{\partial z^{l+1}_k}{\partial z^l_j} = \sum_k \delta^{l+1}_k \cdot \frac{\partial z^{l+1}_k}{\partial z^l_j}$. From the definition of $z^{l+1}_k = \sum_j a^l_j w^{l+1}_{jk} + b^{l+1}_j$ we get $\frac{\partial z^{l+1}_k}{\partial z^l_j} = \sigma'(z^l_j) w^{l+1}_{jk}$. The variation of the error at layer l then becomes $\delta^l_j = \sum_k \sigma'(z^l_j) w^{l+1}_{jk} \delta^{l+1}_k$, or in vector form $\delta^l = \sigma'(z^l) \odot \delta^{l+1}(w^{l+1})^\intercal$. Notice the transpose notation $x^\intercal$ that results from the order of the summing in δ^l_j.
- To get the expression for how the error changes with the bias parameter at any layer of the network, we recall that $z^l_j = \sum_k a^{l-1}_k w^l_{kj} + b^l_j$ which means that $\frac{\partial z^l_j}{\partial b^l_j} = 1$ and $\frac{\partial e}{\partial b^l} = \delta^l$.
- To get the expression for how the error changes with the weight parameter at any layer of the network, we get $\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k$ and $\frac{\partial e}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k$. In vector form this becomes $\frac{\partial e}{\partial w^l} = a^{l-1} \cdot \delta^l$.

5.3 Gradient descent

Now we can apply this to the gradient descent algorithm:

1. Feed the network with the input x and compute z^l and a^l at all layers from 1 to L.
2. Compute the output error $\delta^L = (a^L - y) \odot \sigma'(z^L)$.
3. Backpropagate the error from layer L down to the first layer using the value of δ^{l+1} to compute the value of $\delta^l = \sigma'(z^l) \odot \delta^{l+1}(w^{l+1})^\intercal$.
4. Compute the gradients of the error with regard to the parameters w and b at each layer from δ^l as $\frac{\partial e}{\partial b^l} = \delta^l$ and $\frac{\partial e}{\partial w^l} = \delta^l \cdot a^{l-1}$.
5. Update the values of the parameters at each layer with step γ as $b^l = b^l - \frac{\partial e}{\partial b^l}$ and $w^l = w^l - \frac{\partial e}{\partial w^l}$.

5.4 Gradient descent on the same 2-layer network

Take the previous 2-layer network and compute the gradients.

- $\delta^2 = (a^2 - y) \odot \sigma'(z^2)$, [1, 10]
- $\frac{\partial e}{\partial b^2} = \delta^2$, [1,10]
- $\frac{\partial e}{\partial w^2} = (a^1)^\intercal \cdot \delta^2$, [128, 10]
- $\delta^1 = \sigma'(z^1) \odot \delta^2 (w^2)^\intercal$, [1, 128]
- $\frac{\partial e}{\partial b^1} = \delta^1$, [1,128]
- $\frac{\partial e}{\partial w^1} = (a^0)^\intercal \cdot \delta^1$, [784, 128]

5.5 Gradient descent on tensorflow

The gradient descent for the previous 2-layer network translates into tensorflow as follows. Reuse the feedforward code from the previous sections yet make sure to change the shape of x to '[None, 784]' and y to '[None, 10]' to support multiple images. Then specify the backpropagation and gradient descent codes as follows. This code prints the number of correctly identified images after training with 1000 samples, updating gradients at batches of 10 images at a time as a tradeoff between model accuracy and computational efficiency.

```
# define gradients
def sigmaprime(x):
    return tf.multiply(sigma(x), tf.subtract(tf.constant(1.0), sigma(x)))
delta_2 = tf.multiply(tf.subtract(a_2, y), sigmaprime(z_2))
d_b_2 = delta_2
d_w_2 = tf.matmul(tf.transpose(a_1), delta_2)
delta_1 = tf.matmul(delta_2, tf.transpose(w_2))
delta_1 = tf.multiply(sigmaprime(z_1), delta_1)
d_b_1 = delta_1
d_w_1 = tf.matmul(tf.transpose(a_0), delta_1)
```

We can now run the training graph and iterate to find the parameter values that yield a minimum error.

```
# function calls to update parameters
eta = tf.constant(0.5)
step = [
    tf.assign(w_1, tf.subtract(w_1, tf.multiply(eta, d_w_1))),
    tf.assign(b_1, tf.subtract(b_1, tf.multiply(eta,
                               tf.reduce_mean(d_b_1, axis=[0])))),
    tf.assign(w_2, tf.subtract(w_2, tf.multiply(eta, d_w_2))),
    tf.assign(b_2, tf.subtract(b_2, tf.multiply(eta,
                               tf.reduce_mean(d_b_2, axis=[0]))))
]
# load, run, and save
mnist = input_data.read_data_sets("/tf/data/mnist/", one_hot=True)
saver = tf.train.Saver()
with tf.Session() as sess:
```

```
sess.run(tf.global_variables_initializer())
for i in range(10000):
    # load images
    images, classes = mnist.train.next_batch(10)
    sess.run(step, feed_dict = {x: images, y: classes})
    if i % 1000 == 0:
        res = sess.run(acct_res, feed_dict =
                       {x: mnist.test.images[:1000],
                        y : mnist.test.labels[:1000]})
        print res
    save_path = saver.save(sess, "/tf/data/mnist-2-layer-model/model.ckpt")
```

5.6 Making it easier for the AI developer

Defining the gradient graph is hard work that hardly pays off. Tensorflow can perform automatic differentiation[8] and compute the gradients for you. The assumption is that the implementation of each tensorflow operation also implements the gradient of that function. Check what happens with the function $\sigma(z)$ that you defined manually. You can use `tf.gradients` to compute gradients of a variable with respect to another, or the following:

```
cost = tf.multiply(diff, diff)
step = tf.train.GradientDescentOptimizer(0.1).minimize(cost)
```

Because most people working on neural networks will not build their custom layers but rather resort to well known types of layers, cost functions, and optimizers, neural network libraries work on top of tensorflow to provide these well known constructs to developers. One example of this is Keras; see in particular the layers[9], cost functions[10], and optimizers[11]. The following is the 2-layer network training from section 5.5 in Keras[12].

```
from keras.models import Sequential
from keras.layers import Dense
# create model
model = Sequential()
model.add(Dense(128, input_dim=784, activation='sigmoid'))
model.add(Dense(10, activation='sigmoid'))
#compile model
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
# Fit the model
model.fit(X, Y, epochs=1, batch_size=10)
# evaluate the model
scores = model.evaluate(X, Y)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

[8] https://en.wikipedia.org/wiki/Automatic_differentiation
[9] https://keras.io/layers/core/
[10] https://keras.io/losses/
[11] https://keras.io/optimizers/
[12] https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

6 Adversarial samples for inference – Backpropagation

If we assume that the x, y data points are drawn from some specific random distribution and that a neural network has been trained on a set of samples from this distribution, then it is a matter of time until the network misclassifies new x data drawn at random. In this section we explore how to use the gradient to look for adversarial samples – samples that are misclassified – in a more efficient way than drawing random values.[13]

6.1 Load a saved model

If we want to find adversarial samples for a previously trained model we need first to define that model, load it, and test its accuracy. The code for defining the model is the one shown in 4.3 and must be called before loading the saved file.

```
# load images
mnist = input_data.read_data_sets("/tf/data/mnist/", one_hot=True)
images = mnist.test.images[:1000]
classes = mnist.test.labels[:1000]
# run session
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   saver = tf.train.Saver()
   saver.restore(sess, "/tf/data/mnist-2-layer-model/model.ckpt")
    # run session
    with tf.Session() as sess:
        [acct_res_output, a_output] = sess.run([acct_res, a],
                feed_dict={ x : images, y : classes })
        print(acct_res_output, a_output)
```

6.2 Impact of noise on accuracy

Can't I just generate some noise, add it to the original image, and trick the classifier? Well, supposedly the classifier should handle some level of noise. Let's see.

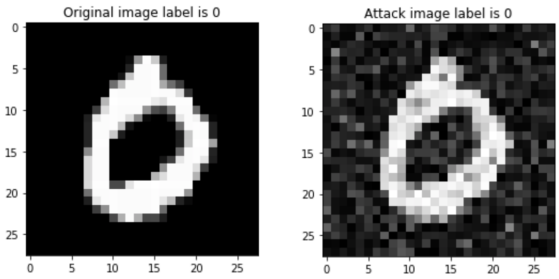
```
x_noise = tf.truncated_normal([1, 784])
eta = tf.constant(0.05)
x_star = tf.clip_by_value(tf.add(x,
                   tf.multiply(eta, tf.sign(x_noise))), 0.0, 1.0)

# run session
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

[13] https://cv-tricks.com/how-to/breaking-deep-learning-with-adversarial-examples-using-tensorflow/

```
saver = tf.train.Saver()
saver.restore(sess, "/tf/data/mnist-2-layer-model/model.ckpt")
x_star_value = images
y_star_value = sess.run(a, feed_dict={ x : x_star_value})
print (np.argmax(y_star_value, axis=1), np.argmax(classes, axis = 1))
for i in range(300):
    x_star_value = sess.run(x_star, feed_dict={ x : x_star_value,
                   y : classes })
    y_star_value = sess.run(a, feed_dict={ x : x_star_value})
    if np.argmax(classes, axis = 1) != np.argmax(y_star_value, axis=1):
        print ('Changed label after {i} iterations'.format(i=i))
        break
plt.title('Original label is {label}'.format(label=np.argmax(classes)))
plt.imshow(images.reshape(28, 28), cmap='gray')
plt.show()
plt.title('Attack label is {label}'.format(label=np.argmax(y_star_value)))
plt.imshow(x_star_value.reshape(28, 28), cmap='gray')
plt.show()
```

Figure 6.2 shows an example of random noise being added to a figure. In this case, the attack image is still classified correctly by the neural network. In order to get more significant results and to better understand the impact of noise a larger set of images should be used. We could find other cases where random noise does change the classification and works as an attack.

Fig. 2. Example of an MNIST random image.



6.3 Gradient descent on the input data x

If we find the gradient of the error with regard to the input x then we can navigate through the input values until we find another value of x^* that is classified wrongly. If the changes to the input value are small then in the case of high-dimensionality input space such as with images, a human observer would find it hard to distinguish normal from adversarial samples.

The gradient of the error with respect to the input data can be obtained with backpropagation in the same way as we found the gradient of the error with respect of the parameters of the neural network. Recall that $z_j^l = \sum_k a_k^{l-1} w_{kj}^l + b_j^l$ and get $\frac{\partial z_j^l}{\partial a_k^{l-1}} = w_{kj}^l$. Then obtain $\frac{\partial e}{\partial a_k^{l-1}} = \delta_j^l \frac{\partial z_j^l}{\partial a_k^{l-1}} = \delta_j^l w_{kj}^l$, which in vector format becomes $\frac{\delta e}{\delta a^{l-1}} = w^l \cdot \delta^l$. For the input data x we get $\frac{\delta e}{\delta x} = \frac{\delta e}{\delta a_0} = w^1 \cdot \delta^1$.

With this gradient we can now run the training graph and replace the parameter updates with the updates of the input data x. An untargeted attack on x yields an x^* that the network classifies with any value except the true value y_{true} for x. The error in this case is the same, i.e. with regard to the y_{true} value as $e_{y_{true}} = \frac{1}{2}(y_{true} - a)^2$ and the gradient 'descent' equation in this case is $x^* = x^* + \gamma \frac{\partial}{\partial x}(e_{y_{true}})$. Notice the + sign before ϵ, which means the gradient method tries to find x^* that causes the largest error and should actually be called gradient ascent. A targeted attack on x yields an x^* that gets classified as y_{target}. For a targeted attack the error is $e_{y_{target}} = \frac{1}{2}(y_{target} - a)^2$, and the gradient descent equation becomes $x^* = x^* - \gamma \frac{\partial}{\partial x}(e_{y_{target}})$. A well known method called Fast Gradient Sign Method uses the sign of the gradient in these equations instead of the actual gradient.

6.4 Gradient descent on x using Tensorflow

The gradient we need is a function of δ^1 for which we already have a computational graph in section 5.5. The following code is for the rest of the graph for the gradient of the error with respect to the input data x and the updated variable x^* (`x_star`) that contains the adversarial sample.

```
d_a_0 = tf.matmul(delta_1, tf.transpose(w_1))
eta = tf.constant(0.05)
x_star = tf.clip_by_value(tf.add(x,
                    tf.multiply(eta, tf.sign(d_a_0))), 0.0, 1.0)
```

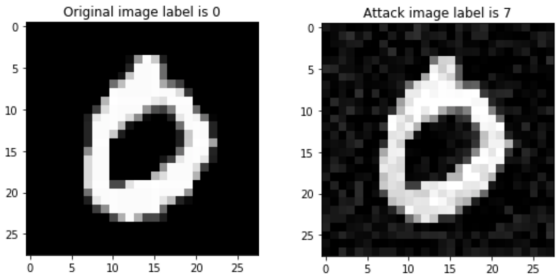
We can take the gradient graph and use it sequentially on an original image to make small changes on the image until we find a new image that the neural network classifies differently than the original image. Hopefully the amount of changes will be small and a human looking at the adversarial image will still recognize the same digit on the adversarial image while the neural network will not. Reuse the code from section 6.2 but change `x_star` to get adversarial samples using the gradient graph.

6.5 Increasing model robustness by including adversarial samples

Run the inference on legitimate samples and on adversarial samples and check their accuracy. Now run the learning algorithm again and include the adversarial samples. How did the accuracy evolve?

Can you now generate new adversarial samples for the new learned model, and see how this will converge after a number of iterations?

Fig. 3. Example of an MNIST adversarial image.



What is the impact of `eta` in the similarity of the adversarial samples to the original samples?

7 CleverHans

Cleverhans [7] is a library that implements many different adversarial learning techniques for inference. You can explore the library here[14].

Copy this tutorial (https://github.com/tensorflow/cleverhans/blob/master/cleverhans_tutorials/mnist_tutorial_keras_tf.py) to the notebook. If necessary consider the following changes.

```
#from cleverhans.compat import flags
from tensorflow.python.platform import flags
...
#check_installation(__file__)
```

Run the example and observe the following. First stage is to train the model. Second stage is to obtain adversarial examples. Check low test accuracy on adversarial examples (0.0402). Third stage is to learn a new model with adversarial samples.

```
Using TensorFlow backend.
num_devices:  1
[INFO 2019-02-12 18:40:04,083 cleverhans] Epoch 0 took 15.09171462059021 seconds
Test accuracy on legitimate examples: 0.9876
...
[INFO 2019-02-12 18:41:22,936 cleverhans] Epoch 5 took 15.069540977478027 seconds
Test accuracy on legitimate examples: 0.9932
Test accuracy on adversarial examples: 0.0402

Repeating the process, using adversarial training
num_devices:  1
```

[14] https://github.com/tensorflow/cleverhans

```
[INFO 2019-02-12 18:42:00,713 cleverhans] Epoch 0 took 34.115516901016235 seconds
Test accuracy on legitimate examples: 0.9821
Test accuracy on adversarial examples: 0.8389
...
[INFO 2019-02-12 18:45:06,274 cleverhans] Epoch 5 took 33.98694682121277 seconds
Test accuracy on legitimate examples: 0.9907
Test accuracy on adversarial examples: 0.9158
```

Challenge: try other evasion techniques in CleverHans.

8 Outlook

8.1 Adversarial AI beyond adversarial samples for inference

The evasion attacks that work at the inference level and that we tried to illustrate here are just one approach at adversarial AI; the following is a quick pointer list to other approaches. Poisoning attacks [5] try to find samples that could be fed to the learning to reduce the performance of the inference. Generative Adversarial Networks [2] lay the groundwork for unsupervised learning with generator and discriminator networks each trying to learn from the output of the others. GAN is a blackbox approach at finding adversarial samples from the output of a discriminator; if in the context of training the discriminator it does makes sense to have an unlimited number of samples from which to train the generator, in the context of an actual attack we may be restricted by rate limits to querying the discriminator [10]. We may want to infer the network parameters but also check if a given sample was used in the training via querying the API which may have privacy implications [9], or provide a watermarking-like mechanism to check if a given neural network-based service has been stolen [16]. We may also be interested in reverse engineering the neural network from binary [15] or hardware [4].

8.2 Adversarial learning in cybersecurity AI

As with any application domain, but especially more given the undeniably adversarial nature of players, we should be especially careful with adversarial attacks to AI used in cybersecurity. Refer to the following for examples of adversarial learning in: malicious URL detection [12], malware domain generation algorithm detection [14], malware traffic detection [8], malware binary detection [3], and cognitive radio jamming [11].

References

1. Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

2. Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat]* (June 2014). arXiv: 1406.2661.
3. Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. Adversarial Perturbations Against Deep Neural Networks for Malware Classification. *arXiv:1606.04435 [cs]* (June 2016). arXiv: 1606.04435.
4. Hua, W., Zhang, Z., and Suh, G. E. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks. In *Proceedings of the 55th Annual Design Automation Conference* (New York, NY, USA, 2018), DAC '18, ACM, pp. 4:1–4:6.
5. Jagielski, M., Oprea, A., Biggio, B., Liu, C., Nita-Rotaru, C., and Li, B. Manipulating Machine Learning: Poisoning Attacks and Countermeasures for Regression Learning. *arXiv:1804.00308 [cs]* (Apr. 2018). arXiv: 1804.00308.
6. Joseph, A. D., Nelson, ., Rubinstein, . I. P., and Tygar, J. D. *Adversarial Machine Learning*. Cambridge University Press, 2019.
7. Papernot, N., Faghri, F., Carlini, N., Goodfellow, I., Feinman, R., Kurakin, A., Xie, C., Sharma, Y., Brown, T., Roy, A., Matyasko, A., Behzadan, V., Hambardzumyan, K., Zhang, Z., Juang, Y.-L., Li, Z., Sheatsley, R., Garg, A., Uesato, J., Gierke, W., Dong, Y., Berthelot, D., Hendricks, P., Rauber, J., and Long, R. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768* (2018).
8. Rigaki, M., and Garcia, S. Bringing a GAN to a Knife-Fight: Adapting Malware Communication to Avoid Detection. In *2018 IEEE Security and Privacy Workshops (SPW)* (May 2018), pp. 70–75.
9. Salem, A., Zhang, Y., Humbert, M., Fritz, M., and Backes, M. ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models. *arXiv:1806.01246 [cs]* (June 2018). arXiv: 1806.01246.
10. Shi, Y., Sagduyu, Y. E., Davaslioglu, K., and Li, J. H. Active Deep Learning Attacks under Strict Rate Limitations for Online API Calls. *arXiv:1811.01811 [cs, stat]* (Nov. 2018). arXiv: 1811.01811.
11. Shi, Y., Sagduyu, Y. E., Erpek, T., Davaslioglu, K., Lu, Z., and Li, J. H. Adversarial Deep Learning for Cognitive Radio Security: Jamming Attack and Defense Strategies. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)* (May 2018), pp. 1–6.
12. Trevisan, M., and Drago, I. Robust URL Classification With Generative Adversarial Networks. *SIGMETRICS Perform. Eval. Rev. 46*, 3 (Jan. 2019), 143–146.
13. Vorobeychik, Y., and Kantarcioglu, M. *Adversarial Machine Learning*. Morgan & Claypool, 2018.
14. Woodbridge, J., Anderson, H. S., Ahuja, A., and Grant, D. Predicting Domain Generation Algorithms with Long Short-Term Memory Networks. *arXiv:1611.00791 [cs]* (Nov. 2016). arXiv: 1611.00791.
15. Xu, M., Liu, J., Liu, Y., Lin, F. X., Liu, Y., and Liu, X. A First Look at Deep Learning Apps on Smartphones. *arXiv:1812.05448 [cs]* (Nov. 2018). arXiv: 1812.05448.
16. Zhang, J., Gu, Z., Jang, J., Wu, H., Stoecklin, M. P., Huang, H., and Molloy, I. Protecting Intellectual Property of Deep Neural Networks with Watermarking. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (New York, NY, USA, 2018), ASIACCS '18, ACM, pp. 159–172.