AI Security Project Report
Adversarial Attack based on White-Box And the Defense Mechanisms for the Adversarial setting

Huang yucheng Huo yicheng

June 14, 2022

Date Performed: February 13, 2022
Instructor: Professor Wang zhibo

1 Experiment Objective

Our group is aiming to reproduce the adversarial attack oriented to Black-Box model and the robust model which is against the attack. Furthermore we are trying to improve both the approach of the way we atttack the model using other loss function and shorten the training time by applying the GPU on the training progess. Last we will test the performance that attack approach show on the robust model we reproduced.

2 Experiment Environment

Here we list the experiment environment in which we run our experiment.

2.1 Dataset

1) MNIST data set from tensorflow.example.tutorials.mnist.
2) CIFAR10 data set. (optional)

2.2 model

Self-build model reproduced from the paper with two Convolutional Neural Network layers and one fully connected layer Towards Deep Learning Models Resistant to Adversarial Attacks.

2.3 Tools and Program Language

Tools	Anaconda
Program Language	Python

2.4 Computing Resources

CPU	Intel(R) Core(TM) i7-9750H CPU @ 2.6GHz
GPU	NVIDIA GeForce GTX 1650

2.5 Software and Libaraies

Cuda Version	V10.0
CuDNN	cudnn-10.1-windows10-x64-v8.0.2.39

3 Scheme Design

3.1 How did the problem raise

AI have had a rapid improvement recently, especially in the region of classifying. Meanwhile because a lot of systems with trained neural network aiming to do classfication work are under use, the problem of security is put into the center of attention. Recent work shows that an adversary is often able to manipulate the input so that the model produces an incorrect output.

3.2 Guiding Idea of our Project

A very small changes to the input image can fool the state-of-art neural networks with high confidence, which gives us an idea about how to ceate adversarial examples. And the impact of network architecture on adversarial robustness is that the capacity plays an important role here. In order to withstand strong adversarial attacks, networks require a larger capacity that for correctly classifying benign examples only.
Building on the above insight, we train networks on MNIST that are robust to a wide range of adversarial attacks. In particual the MNIST network is even robust against white-box attack of an iterative adversary. (But this is not our focus because our testing attack is based on the Black-box)

3.3 Main target of our Project

a. *Reproduce the previous works*. We reproduec the result from paper Towards Deep Learning Models Resistant to Adversarial Attacks. both in attacking approach and the defense approach. Also we reproduce the model itself

b. *Run it to see the result*. Train the model and use the adversarial examples to attack the model to see their performance.

c. *Hardware acceleration*. The origin model training code was running with cpu which is need a huge amount of time. Our group rewrite the code using the GPU and accelerate the training speed for many times.

3.4 Design Requirement

a. *Combine what we learned from class*. Though it is hard but we manage to put what we have learned into practice with hundreds of codes in Python with tutorial in the repository at minst challenge

b. *Compare the attack result with other adversarial examples*. There is a rank table from the author's repository, we are going to compare our result with them to see if the targeted attack will fetch a better success rate.

3.5 Feasibility analysis

a. *Hard to reach the level from origin author*. Due to a quite good performance from the robust model the origin author built, it is difficult to get to the same level as they do.

b. *Lack of computing resources*. As the model training require a lot of computing resources, it might be a little hard for us to run the whole result we previous set due to we have limited computering resource. Though we will accelerate the process by apply the session on the GPU.

c. *Lack of persuasiveness*. We managed to apply our code only to the certain model build by us, so whether it performs well on migration is still in doubt.

4 Scheme Innovation

a. *What's old renew*. Our group not only reproduce the project about training model and attack approach but also use new code and faster method to upgrade it.

b. *What's old brings new*. Our group also apply new attack approach and new way of generate adversarial examples by the method we have learned from the class, which brings us a great challenge.

5 Code Analysis I

There are four main file in our code, we list the important code block below to illustrate thier function.
1) config.json

```
1  "model_dir": "models/a_very_robust_model",
2
3  "random_seed": 4557077,
4  "total_training_steps": 100000,
```

```
  "num_per_output": 100,
  "num_summary_steps": 100,
  "num_per_checkpoint": 300,
  "training_batch_size": 50,

  "num_eval_examples": 10000,
  "eval_batch_size": 200,
  "eval_on_cpu": true,

  "epsilon": 0.3,
  "k": 40,
  "a": 0.01,
  "random_start": true,
  "loss_func": "xent",
  "store_adv_path": "attack.npy"
```

Here the *random seed* refers the random number generator used to initialize the network weights, *total tranining steps* refers that the total training steps, the *num per output* refers to how many training step does the output corresponding to, and *num per checkpoint* refers to training steps each checkpoint corresponding to. *epsilon* limit the perturb, and k, a are all parameters of the attack method, here we choose the loss function as the origin author does which is *xent* – cross-entropy.

2) model.py

In this file of code we create a robust model used to tolerant the perturbation of the adversarial examples.

```
  @staticmethod
  def _weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
  @staticmethod
  def _bias_variable(shape):
    initial = tf.constant(0.1, shape = shape)
    return tf.Variable(initial)
  @staticmethod
  def _conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
  @staticmethod
  def _max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1,
    2,2,1], padding='SAME')
```

Here we use the *staticmethod* to write the funciton we will use in the latter code, where the funciton is basically from the tensorflow.

```
# First layer of convolution
conv1_weight = self._weight_variable([5, 5, 1, 32])
conv1_bias = self._bias_variable([32])
conv1_h = tf.nn.relu(self._conv2d(self.x_image, conv1_weight) +
    conv1_bias)
pool1_h = self._max_pool_2x2(conv1_h)
```

From this part we can see how the model build its convolution layer, set the weight by $tf.truncated\_normal$ then set the bias by intervalo of 0.1. Then use the relu to be the activation funciton and use the matrix we get to do two by two pooling. The second convolution layer is as the same so we won't show it here.

```
  fully_connected_weight = self._weight_variable([7 * 7 * 64,
    1024])
  fully_connected_bias = self._bias_variable([1024])
  pool2_h_flat = tf.reshape(pool2_h, [-1, 7*7*64])
  fully_connected_h = tf.nn.relu(tf.matmul(pool2_h_flat,
    fully_connected_weight) + fully_connected_bias)
```

The fully connected layer is to do a matrix multiplication with the weight parameter and the pooling result from the second convolution layer, then use the relu activation function to obtain the result.

```
   y_xent = tf.nn.sparse_softmax_cross_entropy_with_logits(labels =
     self.y_input, logits=self.pre_softmax)
   self.xent = tf.reduce_sum(y_xent)
```

Last we use the cross entropy to calculate the loss.

3) PGD.py

In this part of code we create the method of create adversarial examples by PGD scheme, here we list the key function of our code.

```
def perturb(self, x_nat, y, sess):
  if self.rand:
  x = x_nat + np.random.uniform(-self.epsilon, self.epsilon, x_nat.
    shape)
  x = np.clip(x, 0, 1)
  else:
  x = np.copy(x_nat)
  for i in range(self.k):
  grad = sess.run(self.grad, feed_dict={self.model.x_input:x, self.
    model.y_input:y})
  x += self.a * np.sign(grad)
  x = np.clip(x, x_nat - self.epsilon, x_nat + self.epsilon)
  x = np.clip(x, 0, 1)
  return x
```

If the random is true(which is set Eternal True by the config file), we use random function to create perturbation within the interval of ± epsilon, use two *clip* function to cut the output to a standard scope.

4) train.py

Finally we introduce the training process, with the file train.py.

```
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  training_time = 0.0

  for ii in range(total_training_steps):
  x_batch, y_batch = mnist.train.next_batch(batch_size)

  start = timer()
  x_batch_adv = attack.perturb(x_batch, y_batch, sess)
  end = timer()
  training_time += end - start
  nat_dict = {model.x_input: x_batch, model.y_input: y_batch}
  adv_dict = {model.x_input: x_batch_adv, model.y_input: y_batch}

  if ii % num_per_output == 0:
  nat_acc = sess.run(model.accuracy, feed_dict=nat_dict)
  adv_acc = sess.run(model.accuracy, feed_dict=adv_dict)
```

```
print('Step {}:    ({})'.format(ii, datetime.now()))
print('    training nat accuracy {:.4}%'.format(nat_acc * 100))
print('    training adv accuracy {:.4}%'.format(adv_acc * 100))
if ii != 0:
print('    {} examples per second'.format(num_per_output *
  batch_size / training_time))
training_time = 0.0

if ii % num_per_checkpoint == 0:
saver.save(sess,
os.path.join(model_dir, 'checkpoint'),
global_step=global_step)

start = timer()
sess.run(train_step, feed_dict=adv_dict)
end = timer()
training_time += end - start
```

Use the origin input from MNIST and the pertrubed input from PGD.py, the model create different output and prediction, then output them in the terminal.

6 Results and Conclusions I

6.1 Results

As we first reproduce the origin result and outstream the result to the terminal, it is quite uneasy to list all the message but we manage the gather some of the Log information.

```
Step 27300:    (2022-06-08 11:48:19.264213)
training nat accuracy 96.0%
training adv accuracy 78.0%
36.5704628637909 examples per second
```

When training for 27300 steps, the accuracy of the natural dataset MNIST is high enough, but the model against the adversarial examples is not good enough for only about 78%, which isn't robust enough to tolerant the adversarial attack.

```
natural: 96.63%
adversarial: 80.40%
avg nat loss: 0.1263
avg adv loss: 0.5775
Waiting for the next checkpoint ...   (2022-06-08
  11:44:15.264644)   ......................
Checkpoint models/a_very_robust_model\checkpoint-27300,
  evaluating ...   (2022-06-08 11:48:25.721207)
```

From the eval side, we set the total eval examples up to 10000, and from the checkpoint generate by the training process, we can see that even for 27300 steps of training, adversarial accuracy is up to 80%.

```
Step 43500:    (2022-06-08 18:56:40.946555)
training nat accuracy 96.0%
training adv accuracy 82.0%
44.05154817717656 examples per second
```

After 43500 steps of training, despite the high accuracy of the origin dataset(because the MNIST is a quite simple dataset to be classified), the adversarial example prediction isn't good enough as we image at the very first beginning. The accuracy seems to be stuck in the nearly 80% around.

```
natural: 97.70%
adversarial: 87.23%
avg nat loss: 0.0737
avg adv loss: 0.3809
Waiting for the next checkpoint ...   (2022-06-08
  18:53:17.261522)   ..................
Checkpoint models/a_very_robust_model\checkpoint-43500,
  evaluating ...   (2022-06-08 18:56:47.647872
```

From the eval side it can be clearly seen that the total accurate number of correct adversarial examples from the total adversarial examples is up to 87%, which is surprisingly better than it performs on the training process.

```
Step 73800:    (2022-06-09 09:20:17.813290)
training nat accuracy 98.0%
training adv accuracy 94.0%
38.82966416796784 examples per second
```

From this time the training process has passed three quarters, and the accuracy of the adversarial examples has reached a satisfying level which can be said to be a robust model for adversarial examples for interval within ϵ of 0.3

```
natural: 98.31%
adversarial: 89.80%
avg nat loss: 0.0527
avg adv loss: 0.2941
Waiting for the next checkpoint ...   (2022-06-09
  09:16:23.814569)   ......................
Checkpoint models/a_very_robust_model\checkpoint-73800,
  evaluating ...   (2022-06-09 09:20:24.178603)
```

It shows the almost same result from the eval side.

```
Step 99900:    (2022-06-09 20:44:02.171968)
training nat accuracy 98.0%
training adv accuracy 90.0%
43.815024366073985 examples per second
```

```
natural: 98.47%
adversarial: 89.53%
avg nat loss: 0.0448
avg adv loss: 0.3077
Waiting for the next checkpoint ...   (2022-06-09
  20:48:17.928656)
```

After the whole process of training the model has showed no obvious changes in the rate of accuracy, so it can be inferred that we may narrow the training step but get the same result.

6.2 Analysis

It can be seen in the Log information that the whole training result is quite good for the MNIST dataset itself for nearly reach the 100% accuracy. As we put out the

scheme of add little perturb in the origin image it indeed acheive the goal of lower the accuracy of the prediction of the model.
For 100000 steps of training our model has finally reach a total 90% average accuracy for the adversarial examples, which can be said to be a robust model that can afford the perturbation for ϵ of ± 0.3.
As for the limitation of computering resources we own, the total training process is up to 74 hours, which is about 3 days, knowing that the training steps after 74000 has made no obvious progress, we decide to reduce the steps of training to decrease the computing time. Also, if we can use the GPU to accelerate we can also make the training process faster.

7 Code Analysis II

We first accelerate the whole program by using the hardware, where GPU support. As the process of optimize the code it is a big challenge for us to modify the code as the tensorflow packet is updating to version 2.x. Here we list the code part we modify the most.

```
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

Use this code to reduce the warning as pre-runnig the code, beacuse there are a huge number of changes as python updating its version.

```
global_step = tf.compat.v1.train.get_or_create_global_step()
train_step = tf.compat.v1.train.AdamOptimizer(1e-4).minimize(
  model.xent, global_step=global_step)

self.x_input = tf.compat.v1.placeholder(tf.float32, shape=[None,
  784])
self.y_input = tf.compat.v1.placeholder(tf.int64, shape=[None])
```

Change the code to match the version at tensorflow 2.x, as tensorflow some how keep the old function using the *compat.v1*.

```
with tf.Session(config=tf.ConfigProto(log_device_placement=True))
   as sess:
```

Change all the session with *log_device_placement* to support GPU(omit other silimar change in other code file).

8 Results and Conclusions II

As we first run the code we aim to test how hardware can accelerate the whole program.

```
Step 1000:    (2022-06-13 15:04:21.558666)
training nat accuracy 74.0%
training adv accuracy 2.0%
237.21543654023898 examples per second
Step 1100:    (2022-06-13 15:04:43.332313)
training nat accuracy 76.0%
training adv accuracy 6.0%
230.8185748090493 examples per second
```

It can be see from the examples per second the code confirm, the whole speed is accelerated up to 4 10 times.

```
Step 15600:    (2022-06-13 16:48:55.074258)
training nat accuracy 98.0%
training adv accuracy 52.0%
239.24254737691675 examples per second
Step 15700:    (2022-06-13 16:49:56.852863)
training nat accuracy 94.0%
training adv accuracy 42.0%
81.87024693377435 examples per second
```

After about a quarter of training, it can be seen that the model has achieve a high accuracy in predicting the origin image in MNIST and about 40% in the accuracy in adversarial examples.

```
natural: 95.93%
adversarial: 52.14%
avg nat loss: 0.2592
avg adv loss: 1.3339
Waiting for the next checkpoint ...   (2022-06-13 16:50:35.344857)
      .
```

But from the eval side, from the whole 10000 adversarial examples the model only reach a poor accuracy against adversarial examples.

```
  Step 49900:    (2022-06-13 21:05:35.157918)
  training nat accuracy 100.0%
  training adv accuracy 94.0%
  120.05093444211309 examples per second
  Step 50000:    (2022-06-13 21:06:30.693362)
  training nat accuracy 100.0%
  training adv accuracy 94.0%
  90.04440934443765 examples per second
```

When training process achieve the half part, both accuracy is up to a quite good level.

```
  natural: 97.86%
  adversarial: 91.79%
  avg nat loss: 0.0659
  avg adv loss: 0.2519
  Waiting for the next checkpoint ...   (2022-06-13
    21:10:25.639418)   .
```

From the eval side, it can also admit that it has already been a good model to tolerant adversarial example.

```
  Step 99800:    (2022-06-14 02:41:48.130489)
  training nat accuracy 100.0%
  training adv accuracy 98.0%
  106.6556946114252 examples per second
  Step 99900:    (2022-06-14 02:42:08.372805)
  training nat accuracy 100.0%
  training adv accuracy 94.0%
  247.10671885746848 examples per second
```

As the end of training, it can be seen that the model reach a very high level of being against the adversarial examples.

```
natural: 98.55%
adversarial: 92.72%
avg nat loss: 0.0427
avg adv loss: 0.2109
```

It can say clearly that this model is a robust enough model for adversarial examples.

8.1 Analysis

From the whole training time we can see a obvious acceleration from the GPU support, the whole training time is reduced to about 20 hours, which can be seen as the five times of acceleration.
Also, there might be some optimization in the function of *compat.v1* because it is clear that the training effect is way better than the first version of code.

9 Thinking and Future Progess

What comes to our mind first is that if a model is built to be robust, then its tolerance against adversarial examples becomes higher. But due to the limitation of the ϵ of the perturbation, the model might not so robust against all the degree of adversarial examples. As we add the ϵ to ± 1 the model accuracy is rapidly dropping to a very low level.
This gives us a thought about advance the scheme of generating adversarial examples, which will definately lower the accuracy of the prediction of the model. But due to the time and the computing power limitation we haven't run the result.
This project gives us a fully review about what is adversarial examples and how to build a model against it in practice. Also the process of improve the code and its performance gives us the perspective that the attack and defense is progessing mutually, there is no limitation to the scheme of attacking and there is no ceiling of defense method either.

10 Appendix

There are four file in our project, here we list a breif introduction of the file.

Bib	the paper we refered
Code	Where we store our codes
Results	Where we store parts of the result we trained
Report	Where we store the report

For code part, we list how to run the code.
1) use *python train.py* to start the train process
2) use *python eval.py* to start the monitor of accuracy from the eval side. You can do this at the same time.
3) you can use the *conda env create -f AISec.yaml* to rebuid the same enviroment as us in anaconda.