Adversarial Attack Report

181220076 周韧哲

1

Nowadays, deep learning has made many outstanding achievements in many fields. Its powerful learning ability makes it the first choice to solve related problems. Neural network is very powerful, training a neural network can have a high accuracy in a certain problem, but deep neural networks are also very fragile.

In 2013, Google researchers pointed out two interesting features of neural networks in a paper entitled <Intriguing properties of neural networks>. First they suggest that it is the space, rather than the individual units, that contains the semantic information in the high layers of neural networks. Second they find that deep neural networks learn input-output mappings that are fairly discontinuous to a significant extent.[5] Therefore, we can see the possibility of deceiving deep learning models by modifying pictures appropriately. The neural network model may be misled by adding slight perturbation (which is not easily perceived by human eyes) into the image that can be correctly classified, so that the wrong results of classification can be obtained. This carefully adjusted image that misleads the neural network model is called Adversarial Example.

With only minor perturbations on a single image, the image can be misclassified with high confidence, or even be classified into a targeted label (not the correct label of the image). This is undoubtedly harmful to the image classification system. For example, a deep neural network in unmanned supermarkets identifies a commodity into another commodity, which will cause customers to have trouble in shopping settlement; In military operations, unmanned aerial vehicles (UAVs) identify people as attackable targets, which can cause tremendous damage to people's lives. At the same time, the study of adversarial examples can make us know more about the deep neural network and further understand the black box of deep learning. That's why we should study adversarial examples.

2

The motivation of attack method is to produce small perturbations to image, which will result in classification error of a neural network classifier. The basic idea is to make the distance between the image and the adversarial image as small as possible and at the same time make the difference of the output result in output space as large as possible. That is to say:

$$\arg\min D(x^*,\mathrm{x}) - \mathrm{L}(\mathrm{f}(x^*),\mathrm{t}) \qquad x^* \in [0,1]$$

Adversarial attack has two categories according to whether the model is known: in white box attack the model f() is known to us; in black box attack the model f() is unknown to us. It also has two categories according to the output result: in untargeted attack the adversarial output just needs to be different from the original output; in targeted attack the adversarial output should be close to the predefined output.

The objective function L() can be cross entropy between $\mathrm{f}(x^*)$ and t or other indicator functions. The common attack methods include Fast Gradient Sign Method (FGSM), Iterative Fast Gradient Sign Method (I-FGSM), Carlini & Wagner Method(C&W) and Zero-Order

Optimization (ZOO).

The solution of FGSM is to maximizing cross entropy between $f(x^*)$ and t subject to $||x^* - x||_\infty \le \varepsilon$ (ε is very small in order to guarantee the perturbation is small), so x^* can be obtained by $x + \varepsilon * sign(\nabla(L(f(x^*),t))$.[1]

IFGSM is iterative FGSM, the updated formula of x^* is $\prod_{x^* \in S}(x + \alpha * sign(\nabla(L(f(x*),t))))$, where α is much smaller than ε and S= $\{x * | [0,1] \cap ||x^* - x||_\infty \le \varepsilon\}$ to guarantee the perturbation is small.[2] We just need to iterate the updated formula for n times to get the adversarial images.

C&W method chooses L2 norm of distance, its purpose is to maximize $L(f(x^*),t)$ and minimize $||x^* - x||_2^2$ so its optimizing objective is

$$\arg\min \lambda||x^* - x||_2^2 - L(f(x^*),t)$$[3]

where λ is a constant. We just need to get the derivative of it so the updated formula of x^* is

$$\prod_{[0,1]}(x^* - \alpha[2\lambda(x^* - x) - \nabla(L(f(x^*),t))])$$

In black box attack, the difficulty is that we can't get the derivative directly, so I use Zero-Order Optimization(ZOO) method to estimates derivatives. In the formula

$$\prod_{[0,1]}(x^* - \alpha[2\lambda(x^* - x) - \nabla(L(f(x*),t))])$$

Where $\nabla(L(f(x*),t)) = \frac{\partial L}{\partial f} * \frac{\partial f}{\partial x^*}$, ZOO method estimates $\frac{\partial f}{\partial x^*}$ as $\frac{f(x^*+h*e)-f(x^*-h*e)}{2h}$ where h is a small number and e is a constant.[4]

The above is the preliminary solutions. Next I will give you my detailed codes to implement the above algorithms.

3

Firstly I trained a CNN model on MNIST.

```
model = Sequential()
model.add(Conv2D(32, (5,5), activation='relu', input_shape=[28, 28, 1]))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Conv2D(64, (5,5), activation='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

The CNN model has two convolutional layers and two pooling layers. In the first convolutional layer the filter's depth is 32 and its size is 5*5, the activation function is relu and the second convolutional layer the filter's depth is 64 with 5*5 size. The pooling layers are max pooling. I add a Flatten layer to one-dimensional the input. In implicit nodes dropout rate is set as 0.5. The first full connected layer's activation function is relu and the second is softmax.

```
model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=1)
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)
```

And then I started to compile the model using AdamOptimizer, finally the test accuracy is up to 0.9858， which is a very excellent result. It shows that this CNN model have a very good performance in the MNIST classification problem.

```
Epoch 1/1
60000/60000 [==============================] - 28s 469us/step - loss: 0.2369 - acc: 0.9254
10000/10000 [==============================] - 2s 154us/step
Test accuracy: 0.9858
```

Next, I'll give you a detailed description of the different attack methods mentioned above.

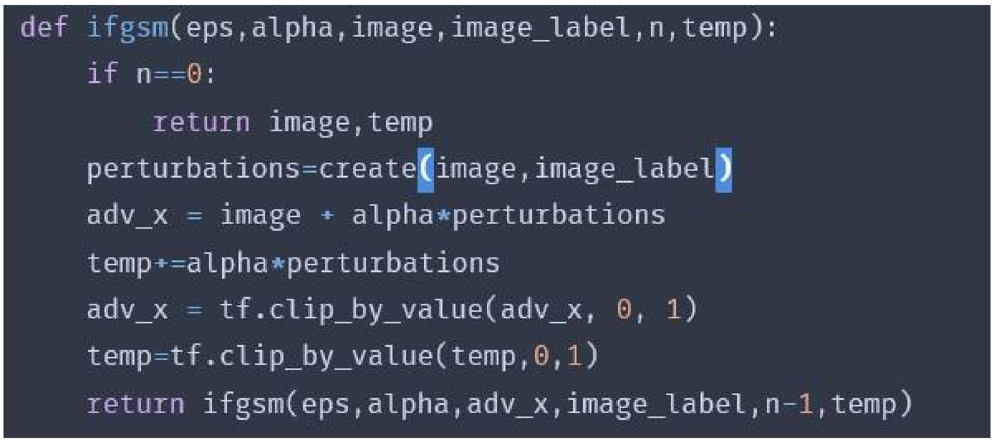
IFGSM method:

By using the updated formula $\prod_{x^* \in S}(x + \alpha * sign(\nabla(L(f(x*),t))))$, we can write IFGSM attack method below.

Function create() is to create perturbations according to the input images and labels. Temp is a size*10 matrix with a probability of 1 at the tag's location and a probability of 0 at other locations. Loss is the cross entropy between prediction and temp. So the perturbations are the loss's gradients towards the input images.

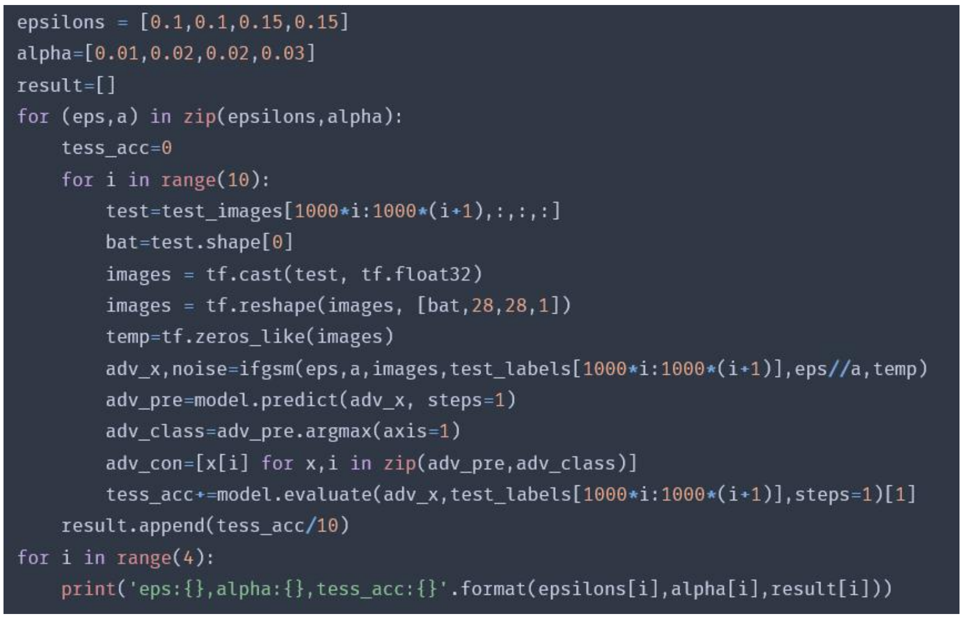
```
def create(input_image,input_label):
    size=input_image.shape[0]
    temp=np.zeros((size,10))
    for i in range(temp.shape[0]):
        temp[i,input_label[i]]=1
    prediction=model(input_image)
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        loss = loss_object(prediction, temp)
  # Get the gradients of the loss w.r.t to the input image.
    gradient = tf.gradients(loss, input_image)
    signed_grad = tf.sign(gradient)
    return tf.reshape(signed_grad,[size,28,28,1])
```

Function ifgsm() is a recursive function. N is the iteration times. Temp is thee noise of the adversarial image.



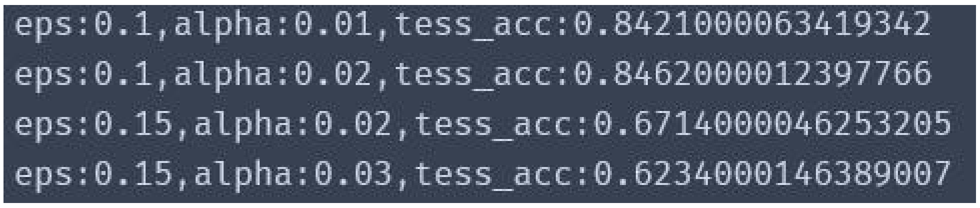
```
def ifgsm(eps,alpha,image,image_label,n,temp):
    if n==0:
        return image,temp
    perturbations=create(image,image_label)
    adv_x = image + alpha*perturbations
    temp+=alpha*perturbations
    adv_x = tf.clip_by_value(adv_x, 0, 1)
    temp=tf.clip_by_value(temp,0,1)
    return ifgsm(eps,alpha,adv_x,image_label,n-1,temp)
```

Because my computer doesn't have enough memory, I can only run 1,000 pictures at a time. So I ran 10,000 test pictures in a loop. Adv_x is the adversarial images, noise is the average L2 norm of adversarial noise on all adversarial images. Adv_pre is the prediction of probability matrix of the adversarial images, adv_class is the predictive label of the adversarial images, and adv_con is the confidence of the predictive label.



```
epsilons = [0.1,0.1,0.15,0.15]
alpha=[0.01,0.02,0.02,0.03]
result=[]
for (eps,a) in zip(epsilons,alpha):
    tess_acc=0
    for i in range(10):
        test=test_images[1000*i:1000*(i+1),:,:,:]
        bat=test.shape[0]
        images = tf.cast(test, tf.float32)
        images = tf.reshape(images, [bat,28,28,1])
        temp=tf.zeros_like(images)
        adv_x,noise=ifgsm(eps,a,images,test_labels[1000*i:1000*(i+1)],eps//a,temp)
        adv_pre=model.predict(adv_x, steps=1)
        adv_class=adv_pre.argmax(axis=1)
        adv_con=[x[i] for x,i in zip(adv_pre,adv_class)]
        tess_acc+=model.evaluate(adv_x,test_labels[1000*i:1000*(i+1)],steps=1)[1]
    result.append(tess_acc/10)
for i in range(4):
    print('eps:{},alpha:{},tess_acc:{}'.format(epsilons[i],alpha[i],result[i]))
```

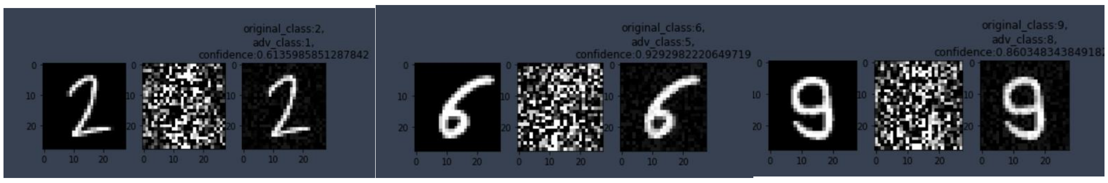
As you can see I try four different kinds of epsilons and alpha, the test accuracy is:



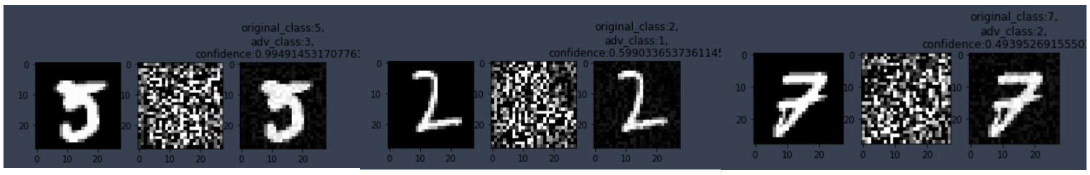
```
eps:0.1,alpha:0.01,tess_acc:0.8421000063419342
eps:0.1,alpha:0.02,tess_acc:0.8462000012397766
eps:0.15,alpha:0.02,tess_acc:0.6714000046253205
eps:0.15,alpha:0.03,tess_acc:0.6234000146389007
```

The accuracy of the model on the test set is reduced to 0.62 with eps=0.15 and alpha=0.03. It can be seen that this attack method has some effect because it reduces the accuracy of the model's prediction by a large margin.

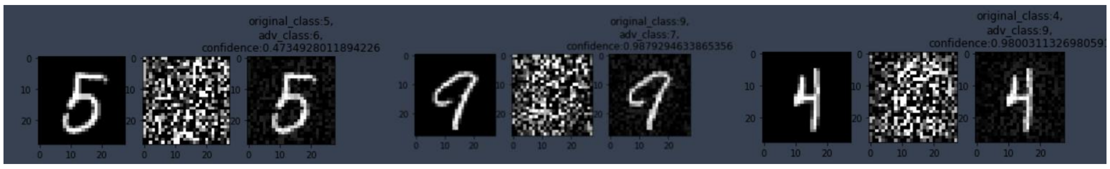
Epsilon=0.1,alpha=0.01, some of the adversarial images:


original_class:2, adv_class:1, confidence:0.6135985851287842
original_class:6, adv_class:5, confidence:0.9292982220649719
original_class:9, adv_class:8, confidence:0.86034834384918

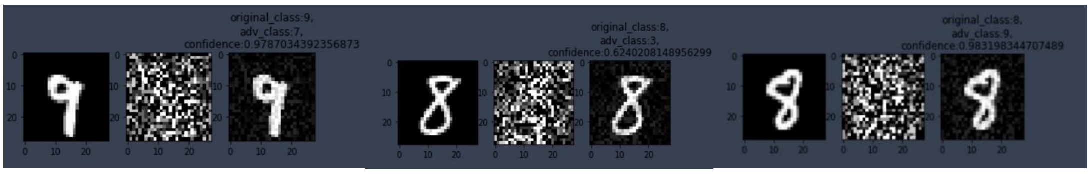
Epsilon=0.1,alpha=0.02, some of the adversarial images:


original_class:5, adv_class:3, confidence:0.99491453170776
original_class:2, adv_class:1, confidence:0.599033653736114
original_class:7, adv_class:2, confidence:0.4939526915550

Epsilon=0.15,alpha=0.02, some of the adversarial images:


original_class:5, adv_class:6, confidence:0.4734928011894226
original_class:9, adv_class:7, confidence:0.9879294633865356
original_class:4, adv_class:9, confidence:0.98003113269805

Epsilon=0.15,alpha=0.03, some of the adversarial image:

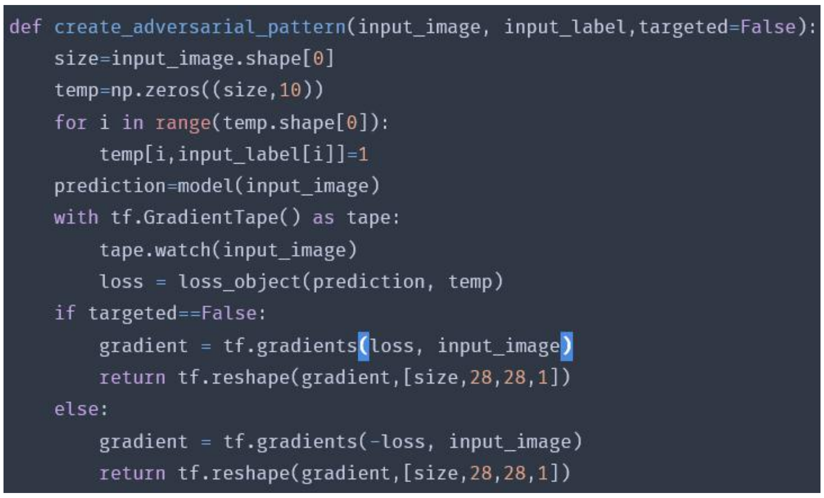

original_class:9, adv_class:7, confidence:0.9787034392356873
original_class:8, adv_class:3, confidence:0.6240208148956299
original_class:8, adv_class:9, confidence:0.983198344707489

We can see that when epsilon=0.1,alpha=0.01 the adversarial images are barely abnormal, and the test accuracy is 0.84, which shows that the effect of these parameters is very good. And when epsilon=0.15, alpha=0.02, the adversarial images have some visible pixel changes which means the attack effect is not very good.

C&W attack（targeted and untargeted）:

By using the updated formula $\prod_{[0,1]}\left(x^{*}-\alpha[2\lambda(x^{*}-x)-\nabla(\mathrm{L}(\mathrm{f}(x^{*}),\mathrm{t}))]\right)$ we can write the targeted and untargeted C&W attack method below.

Function create_adversarial_pattern() is to create perturbations according to the input images and labels. When parameter targeted is False the input_label should be the original labels of the images, when parameter targeted is True the input_label is the assigned labels.



```python
def create_adversarial_pattern(input_image, input_label,targeted=False):
    size=input_image.shape[0]
    temp=np.zeros((size,10))
    for i in range(temp.shape[0]):
        temp[i,input_label[i]]=1
    prediction=model(input_image)
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        loss = loss_object(prediction, temp)
    if targeted==False:
        gradient = tf.gradients(loss, input_image)
        return tf.reshape(gradient,[size,28,28,1])
    else:
        gradient = tf.gradients(-loss, input_image)
        return tf.reshape(gradient,[size,28,28,1])
```

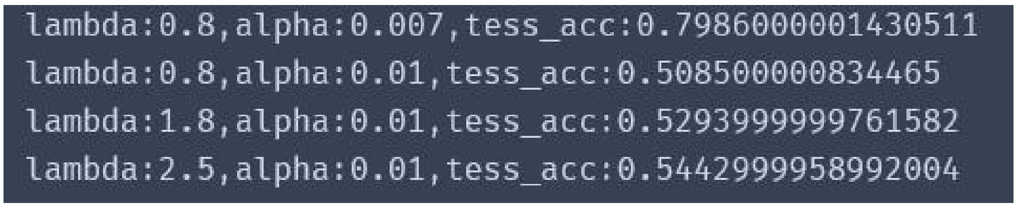
Untargeted C&W attack:
When targeted is False, we just need to maximize the cross entropy between prediction and temp to deviate the prediction of the model from the original labels.
Targeted C&W attack:
When targeted is True, we just need to minimize the cross entropy between prediction and temp which is to say maximize the negative cross entropy between prediction and temp to make the prediction of the model biased to the assigned labels.

```
def c_w(alpha,image,label,maxiter,Lambda,targeted=False):
    image_shadow=image
    temp=tf.zeros_like(image)
    for i in range(maxiter):
        perturbations=create_adversarial_pattern(image,label,targeted=targeted)
        adv_x = image + alpha*perturbations - 2*alpha*Lambda*(image-image_shadow)
        adv_x = tf.clip_by_value(adv_x, 0, 1)
        image_shadow=image
        image=adv_x
        temp+=alpha*perturbations - 2*alpha*Lambda*(image-image_shadow)
        temp=tf.clip_by_value(temp,0,1)
    return image,temp
```

Function c_w() is to generate the adversarial images according to the updated formula $\prod_{[0,1]} (x^* - \alpha[2\lambda(x^* - x) - \nabla(\mathrm{L}(\mathrm{f}(x*),\mathrm{t}))])$, I initialize image_shadow to image, and execute this attack for maxiter times.
Adv_x is the adversarial images, noise is the average L2 norm of adversarial noise on all adversarial images. Adv_pre is the prediction of probability matrix of the adversarial images, adv_class is the predictive label of the adversarial images, and adv_con is the confidence of the predictive label. I try alpha, Lambda for 0.007,0.8 and 0.01,0.8 and 0.01,1.8 and 0.01,2.5.
Untargeted attack:
Firstly I try the untargeted attack.

```
alpha=[0.007,0.01,0.01,0.01]
Lambda=[0.8,0.8,1.8,2.5]
result=[]
for (lam,a) in zip(Lambda,alpha):
    tess_acc=0
    for i in range(10):
        test=test_images[1000*i:1000*(i+1),:,:,:]
        bat=test.shape[0]
        images = tf.cast(test, tf.float32)
        images = tf.reshape(images, [bat,28,28,1])
        adv_x,noise=c_w(a,images,test_labels[1000*i:1000*(i+1)],15,lam,targeted=False)
        adv_pre=model.predict(adv_x, steps=1)
        adv_class=adv_pre.argmax(axis=1)
        adv_con=[x[i] for x,i in zip(adv_pre,adv_class)]
        tess_acc+=model.evaluate(adv_x,test_labels[1000*i:1000*(i+1)],steps=1)[1]
    result.append(tess_acc/10)
for i in range(4):
    print('lambda:{},alpha:{},tess_acc:{}'.format(Lambda[i],alpha[i],result[i]))
```

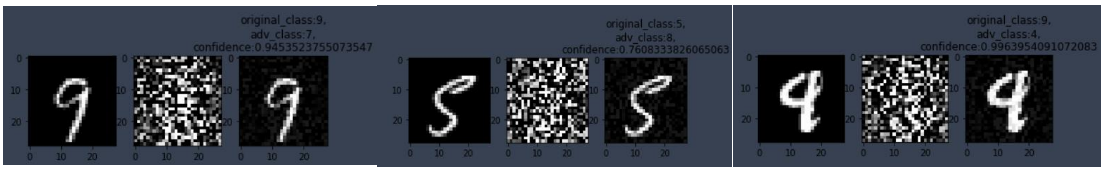
The accuracy on the test set is:



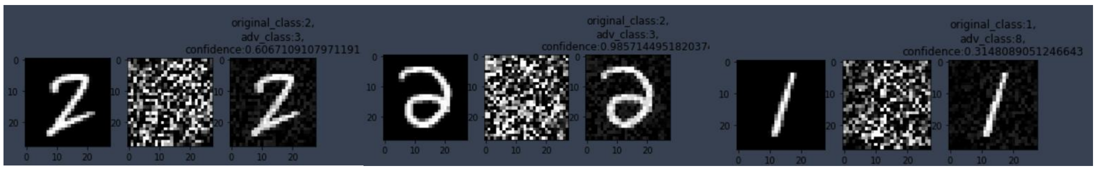
```
lambda:0.8,alpha:0.007,tess_acc:0.7986000001430511
lambda:0.8,alpha:0.01,tess_acc:0.508500000834465
lambda:1.8,alpha:0.01,tess_acc:0.5293999999761582
lambda:2.5,alpha:0.01,tess_acc:0.5442999958992004
```

We can see that this attack method has a better performance than IFGSM. The test accuracy even is reduced to 0.51 with lambda=0.8 and alpha=0.01. it reduces the accuracy of the model's prediction by a large margin.

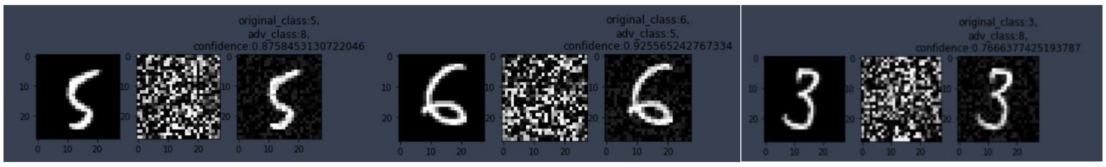
Lambda=0.8, alpha=0.007, some of the adversarial images:


original_class:9,
adv_class:7,
confidence:0.9453523755073547
original_class:5,
adv_class:8,
confidence:0.7608333826065063
original_class:9,
adv_class:4,
confidence:0.9963954091072083

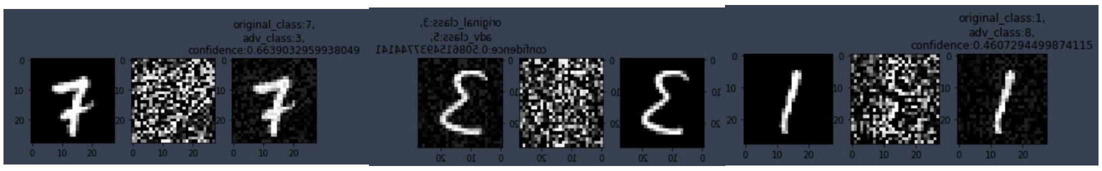
Lambda=0.8, alpha=0.01, some of the adversarial images:



Lambda=1.8, alpha=0.01, some of the adversarial images:



Lambda=2.5, alpha=0.01, some of the adversarial images:

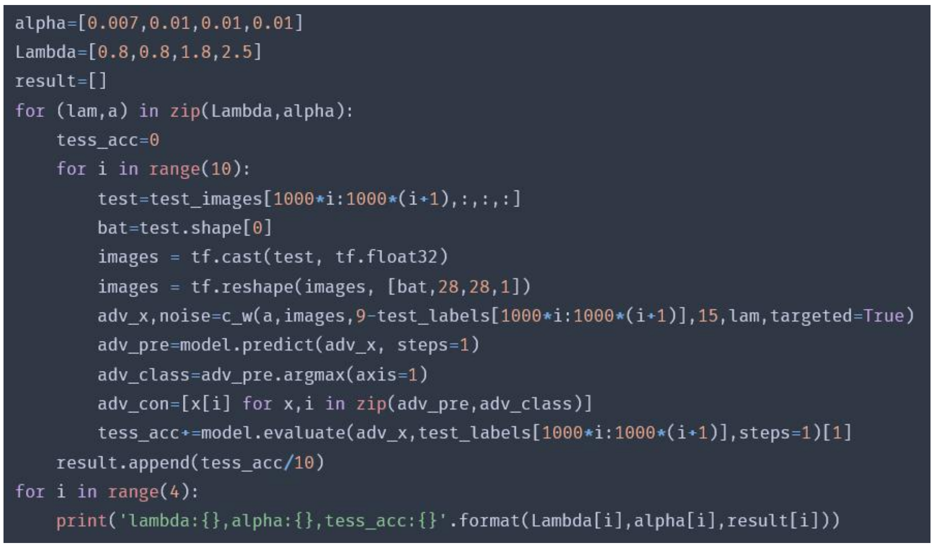

original_class:7,
adv_class:3,
confidence:0.6639032959938049
original_class:3,
adv_class:5,
confidence:0.5086154937744141
original_class:1,
adv_class:8,
confidence:0.4607294499874115

We can see that when Lambda=0.8, alpha=0.007 the perturbations of the adversarial images are almost imperceptible. And when Lambda=2.5, alpha=0.01 we can see some visible pixel changes.

Targeted attack:

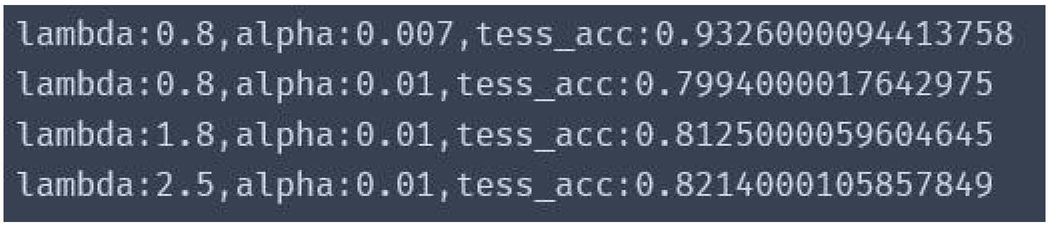
Set the parameter targeted to True, then the attack becomes targeted attack. The input labels should be the assigned label of each image so for convenience I assign the input label as 9- the original label, which means if a label is 1 it should be predicted as 8 and if a label is 4 it should be predicted as 5.

The other setting is similar to the untargeted attack. My purpose is to compare the untargeted attack method and the targeted attack method under the same parameters to figure out the difference.



```
alpha=[0.007,0.01,0.01,0.01]
Lambda=[0.8,0.8,1.8,2.5]
result=[]
for (lam,a) in zip(Lambda,alpha):
    tess_acc=0
    for i in range(10):
        test=test_images[1000*i:1000*(i+1),:,:,:]
        bat=test.shape[0]
        images = tf.cast(test, tf.float32)
        images = tf.reshape(images, [bat,28,28,1])
        adv_x,noise=c_w(a,images,9-test_labels[1000*i:1000*(i+1)],15,lam,targeted=True)
        adv_pre=model.predict(adv_x, steps=1)
        adv_class=adv_pre.argmax(axis=1)
        adv_con=[x[i] for x,i in zip(adv_pre,adv_class)]
        tess_acc+=model.evaluate(adv_x,test_labels[1000*i:1000*(i+1)],steps=1)[1]
    result.append(tess_acc/10)
for i in range(4):
    print('lambda:{},alpha:{},tess_acc:{}'.format(Lambda[i],alpha[i],result[i]))
```

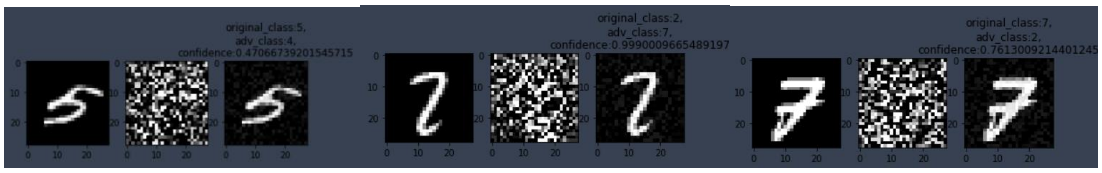
The accuracy on the test set is:



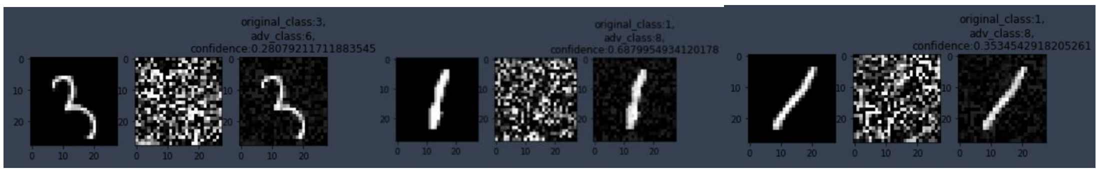
```
lambda:0.8,alpha:0.007,tess_acc:0.9326000094413758
lambda:0.8,alpha:0.01,tess_acc:0.7994000017642975
lambda:1.8,alpha:0.01,tess_acc:0.8125000059604645
lambda:2.5,alpha:0.01,tess_acc:0.8214000105857849
```

It can be seen that targeted attack's effect is worse than untargeted attack's under the same parameters. But it still has some effect which reduces the accuracy of the CNN model's prediction.

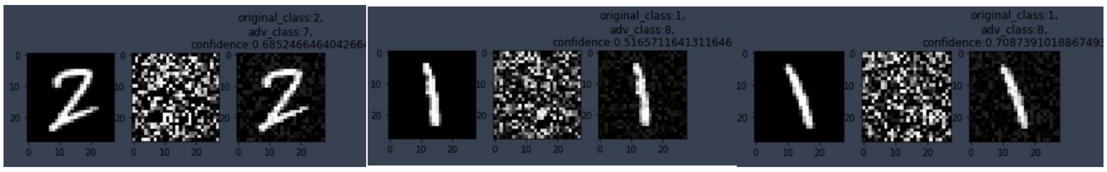
Lambda=0.8, alpha=0.007, some of the adversarial images:



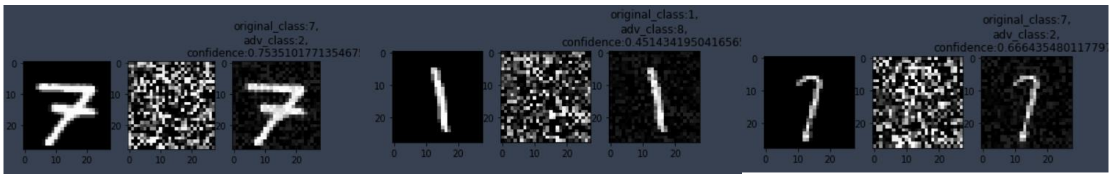
Lambda=0.8, alpha=0.01, some of the adversarial images:



Lambda=1.8, alpha=0.01, some of the adversarial images:



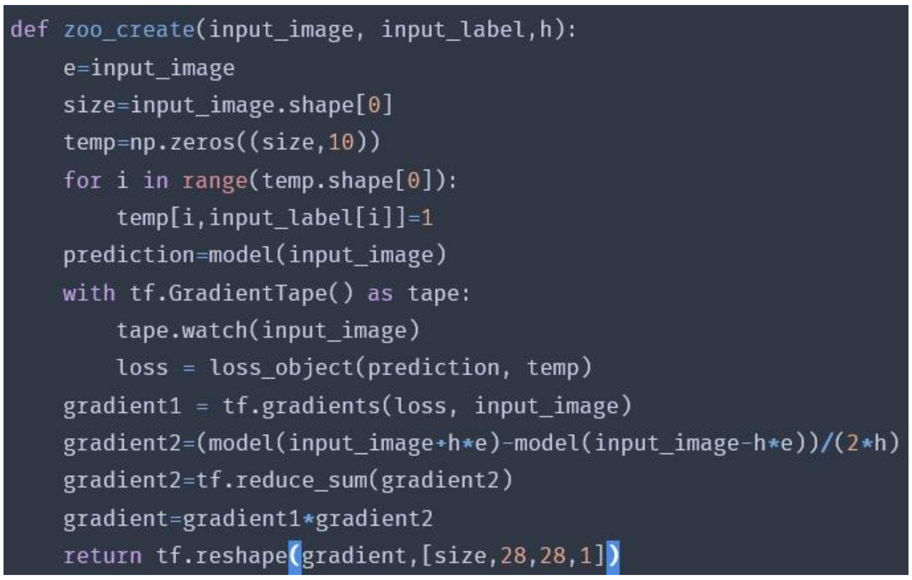
Lambda=2.5, alpha=0.01, some of the adversarial images:



We can see that when Lambda=0.8, alpha=0.007 the perturbations of the adversarial images are almost invisible. And when Lambda=0.8,1.8,2.5, alpha=0.01 we can see some visible pixel changes on the adversarial images.

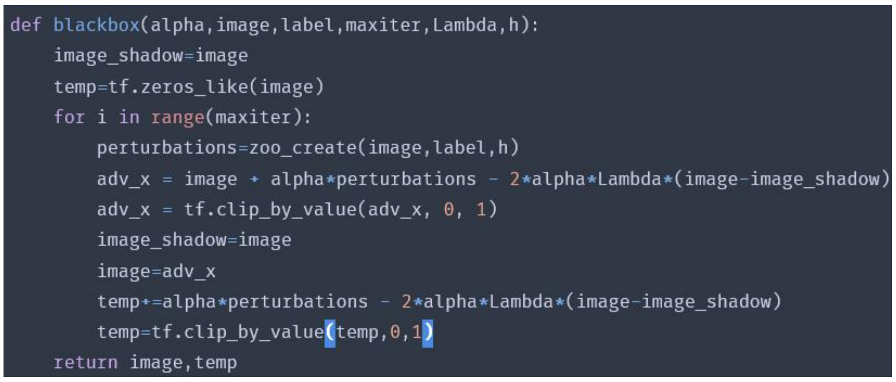
Black-box C&W attack:

By using the updated formula $\prod_{[0,1]} (x^* - \alpha[2\lambda(x^* - x) - \nabla(\mathrm{L}(\mathrm{f}(x *),\mathrm{t}))])$ where $\nabla(\mathrm{L}(\mathrm{f}(x *),\mathrm{t})) = \frac{\partial L}{\partial f} * \frac{\partial f}{\partial x^*}$ and $\frac{\partial f}{\partial x^*}$ is estimated as $\frac{f(x^*+h*e)-f(x^*-h*e)}{2h}$, we can write the Black-box C&W attack below.



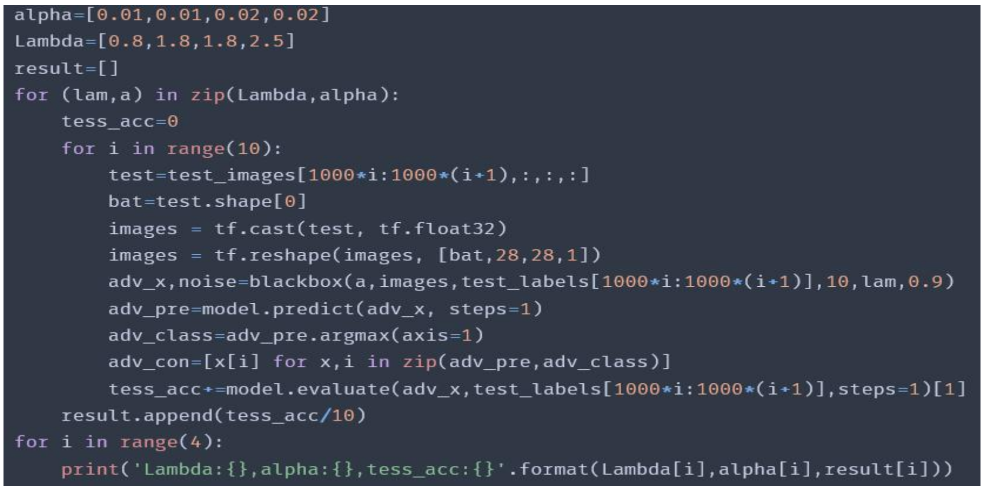
```
def zoo_create(input_image, input_label,h):
    e=input_image
    size=input_image.shape[0]
    temp=np.zeros((size,10))
    for i in range(temp.shape[0]):
        temp[i,input_label[i]]=1
    prediction=model(input_image)
    with tf.GradientTape() as tape:
        tape.watch(input_image)
        loss = loss_object(prediction, temp)
    gradient1 = tf.gradients(loss, input_image)
    gradient2=(model(input_image+h*e)-model(input_image-h*e))/(2*h)
    gradient2=tf.reduce_sum(gradient2)
    gradient=gradient1*gradient2
    return tf.reshape(gradient,[size,28,28,1])
```

Function zoo_create() is to create perturbations according to the input images and labels. I initialize e as input_image. Gradient1 is $\frac{\partial L}{\partial f}$ and gradient2 is $\frac{\partial f}{\partial x^*}$ which is estimated as $\frac{f(x^*+h*e)-f(x^*-h*e)}{2h}$, so the final gradient is gradient1*gradient2 which is the perturbations of the input images.



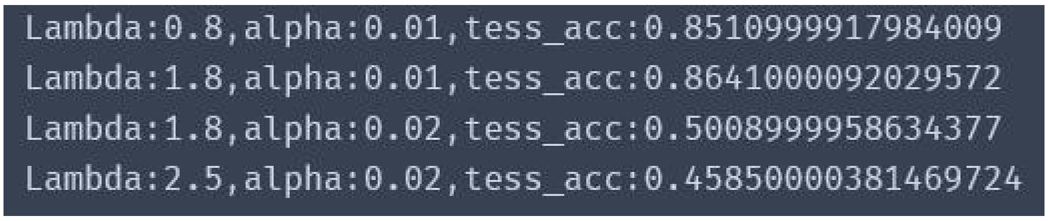
```
def blackbox(alpha,image,label,maxiter,Lambda,h):
    image_shadow=image
    temp=tf.zeros_like(image)
    for i in range(maxiter):
        perturbations=zoo_create(image,label,h)
        adv_x = image + alpha*perturbations - 2*alpha*Lambda*(image-image_shadow)
        adv_x = tf.clip_by_value(adv_x, 0, 1)
        image_shadow=image
        image=adv_x
        temp+=alpha*perturbations - 2*alpha*Lambda*(image-image_shadow)
        temp=tf.clip_by_value(temp,0,1)
    return image,temp
```

Function blackbox() is to generate the adversarial images according to the updated formula $\prod_{[0,1]} (x^* - \alpha[2\lambda(x^* - x) - \nabla(\mathrm{L}(\mathrm{f}(x *),\mathrm{t}))])$, I initialize image_shadow to image, and execute this attack for maxiter times.



```python
alpha=[0.01,0.01,0.02,0.02]
Lambda=[0.8,1.8,1.8,2.5]
result=[]
for (lam,a) in zip(Lambda,alpha):
    tess_acc=0
    for i in range(10):
        test=test_images[1000*i:1000*(i+1),:,:,:]
        bat=test.shape[0]
        images = tf.cast(test, tf.float32)
        images = tf.reshape(images, [bat,28,28,1])
        adv_x,noise=blackbox(a,images,test_labels[1000*i:1000*(i+1)],10,lam,0.9)
        adv_pre=model.predict(adv_x, steps=1)
        adv_class=adv_pre.argmax(axis=1)
        adv_con=[x[i] for x,i in zip(adv_pre,adv_class)]
        tess_acc+=model.evaluate(adv_x,test_labels[1000*i:1000*(i+1)],steps=1)[1]
    result.append(tess_acc/10)
for i in range(4):
    print('Lambda:{},alpha:{},tess_acc:{}'.format(Lambda[i],alpha[i],result[i]))
```

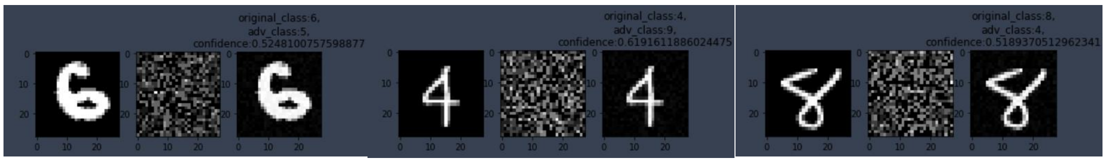
Adv_x is the adversarial images, noise is the average L2 norm of adversarial noise on all adversarial images. Adv_pre is the prediction of probability matrix of the adversarial images, adv_class is the predictive label of the adversarial images, and adv_con is the confidence of the predictive label.I tried alpha, Lambda for 0.01,0.8 and 0.01,1.8 and 0.02,1.8 and 0.02,2.5. The accuracy on the test set is:



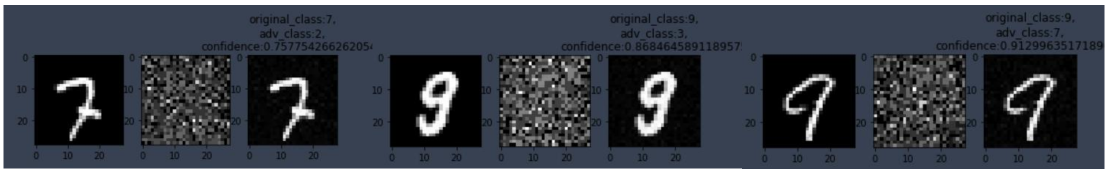
```
Lambda:0.8,alpha:0.01,tess_acc:0.8510999917984009
Lambda:1.8,alpha:0.01,tess_acc:0.8641000092029572
Lambda:1.8,alpha:0.02,tess_acc:0.5008999958634377
Lambda:2.5,alpha:0.02,tess_acc:0.45850000381469724
```

When Lambda=2.5, alpha=0.02 the test accuracy is even reduced to 0.46, which is a very excellent performance.

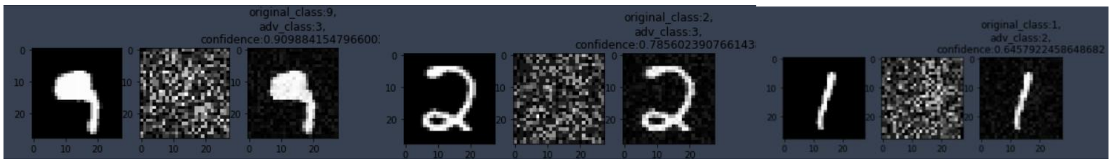
Lambda=0.8, alpha=0.01, some of the adversarial images:


original_class:6,
adv_class:5,
confidence:0.5248100757598877
original_class:4,
adv_class:9,
confidence:0.6191611886024475
original_class:8,
adv_class:4,
confidence:0.5189370512962341

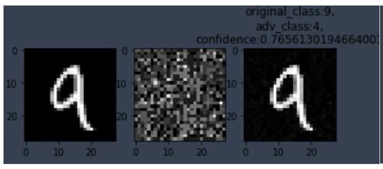
Lambda=1.8, alpha=0.01, some of the adversarial images:

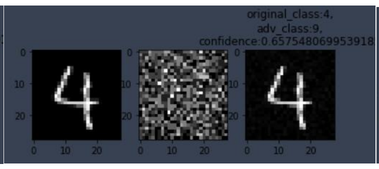


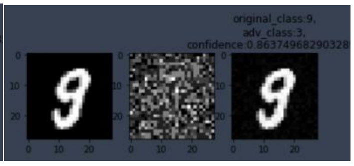
Lambda=1.8, alpha=0.02, some of the adversarial images:



Lambda=2.5, alpha=0.02, some of the adversarial images:







We can see that all the adversarial images above almost have no visible pixel changes, which means they can deceive the neural networks without being discovered by human eyes. So this attack method has achieved excellent results.

In summary, I tried four different attack methods include IFGSM, C&W(targeted and untargeted) and Blackbox C&W. All the attack methods have a certain degree of effect. I figure out that no matter what method the basic idea to create the perturbation is to make the distance between the image and the adversarial image as small as possible and at the same time make the difference of the output result in output space as large as possible.

Reference

[1] Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy, “Explaining and Harnessing Adversarial Examples”, ICLR 2015
[2] Alexey Kurakin, Ian Goodfellow, Samy Bengio, “Adversarial Examples in the Physical World”, Arxiv 2017
[3] Nicholas Carlini, David Wagner, “Towards Evaluating the Robustness of Neural Networks”, Arxiv 2017
[4] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, Cho-Jui Hsieh, “ZOO: Zeroth Order Optimization based Black-box Attacks to Deep Neural Networks without Training Substitute Models”, AISEC 2017
[5] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, Rob Fergus, “Intriguing properties of neural networks”, CVPR 2014