ADVERSARIAL EXAMPLES

INTRODUCTION

In this assignment the goal is to examine the dependency of adversarial examples on networks with different architectures and different initial values. Adversarial examples are on purpose designed in order to fool the prediction of the model. Many of these examples are implemented as small perturbation of the original image. This perturbations are often undistinguishable to human, however they strongly affect the precision of a network and big degradation of network accuracy can be detected.

The main importance of adversarial examples is related to transferability to other networks: adversarial networks generated to fool one model can also fool other models with different architectures. From this consideration, it is clear that adversarial examples topic is fundamental for what concerns security risk and misclassification investigation (for example, consider its importance related to facial recognition application or self-driving cars). Some of the most common technique to defend against adversarial example are Adversarial Training, Autoencoders, Defensive Distillation and DeepSafe.

As seen during class, there are many different techniques to generate adversarial examples. In this assignment, the used one is among the easiest and most effective: Fast Gradent Sign Method. The working principle of this method relies on adding a certain amount of noise (defined by algorithm parameter *Epsilon*) generated from the gradient of the Loss function of the target network. In this way, FGSM generates an adversarial example in order to maximize the probability of fooling the network. Note however that FGSM is designed to be very fast, and does not search for the optimal perturbation.

CODE EXPLANATION

The code used to realize this assignment is base on the provided template. Initially, MNIST dataset is imported (notice that batch size of test_loader must be set to 1 in order to have the code working). Then, a script for testing a network is realized. This function will be called every time that we want to test the effects of generated adversarial examples (for the Target Network) on a different network.

After this initial setup, the four Networks required for this assignment are defined. We use 2 Convolutional Neural Networks with different architectures, but we initialize each of them with 2 different initial weights, in order to have a total of 4 Nets. Used Networks were created in Assignment 1. For the purpose of this assignment, I selected as Network A a CNN with 2 filters and 2 fully connected layers with Adam optimizer and as Network B a CNN with 1 filter, 2 fully connected layers and SGD optimizer. They both have a very good overall accuracy, but one is slightly less precise with respect to the other (99.09 % VS 97.16 %)

NETWORK A:

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=3136, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=10, bias=True)
)
```

NETWORK B:

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (drop_out): Dropout(p=0.5, inplace=False)
  (fc1): Linear(in_features=6272, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=10, bias=True)
)
```

INITIAL ACCURACY OF THE NETWORKS

	NET A	NET B
ACCURACY	99.09 %	97.16 %

As mentioned above, the total Networks are four because also Network A’ and Network B’ are created: they share the same architectures of the other two nets, but they are initialized with different initial weights (or net parameters). This is possible because layers are initialized in some random ways after creation. For example, linear layers are initialized with the following uniform distribution:

```
stdv = 1. / math.sqrt(self.weight.size(1))
self.weight.data.uniform_(-stdv, stdv)
if self.bias is not None:
    self.bias.data.uniform_(-stdv, stdv)
```

and something similar is done for the convolutional layers. Since they are randomly initialized, initialize a Network with the same architecture two times leads to two different networks with different initial parameters, and this is what is done in this assignment. Notice that during training, the optimizer will start from these initial parameters and will update them during the process. At the end of the training, since the starting conditions are different, the two networks will have different parameters. In the consideration section, we will see that network with the same architecture but different initial weights lead to different results when tested against the same adversarial example.

After Networks definition, there is the core of the FGSM algorithm. First, the functions that are called in order to perform an FGSM attack to one net are defined.

In the fgsm_attack function, the perturbed image is created from the formula seen in class:

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$

In the test function, testing with respect to adversarial examples images is performed: for every testing image in the MNIST test dataset (10000) elements, the image is perturbed through fgsm_attack function and is fed to the target network in order to make classification. At the end, from the number of misclassified digits it is possible to compute the overall accuracy of the network with respect to adversarial examples. Notice that in the test function, also 3 different vectors are defined:

1) adv_examples is a vector that contains 5 elements for a later visualization of an example of perturbed digits
2) adv_samples is a vector that contains all the generated adversarial examples. On this vector, the accuracy of the other networks will be tested in order to see the effect on them of the adversarial examples.
3) sample_labels is of course necessary to store the corresponding labels of the adv_samples in order to compute the accuracy of the other networks.

Also, a vector epsilons containing epsilon values for the algorithm is initialized (optional part of the assignment).

After that, 4 cells are defined: in each of them a Net among the four at disposal is chosen as a target for the FSGM attack. Considering as instance the FGSM ATTACK: NET A. A for loop is used in order to produce the attack with different epsilon values. In each iteration, the test function is invoked: it receives as input the Network model, the device and the test_loader set and return as outputs 4 elements (acc, ex, Net_1_adv_samples and sample_labels) according to procedure above described. In particular, acc and ex are the overall accuracy and the adv_examples described above. They are appended to two vectors for later plot of graphs highlighting changes according to epsilon values.

After the attack to the Target Network, dataloader set containing the Net_1_adv_samples and sample_labels is generated in order to be passed later to evaluate_model function. For each of the three remaining networks, evaluate_model function is invoked in order to test accuracy of these networks with respect to adversarial examples. Here is reported an example of the displayed output of this cell:

```
Epsilon: 0.1	Test Accuracy = 9666 / 10000 = 96.66%


Testing the network...
Net_2 Accuracy: 99.14668833807396 %


Testing the network...
Net_3 Accuracy: 96.76960585127998 %


Testing the network...
Net_4 Accuracy: 96.77976432344575 %


Epsilon: 0.2	Test Accuracy = 9597 / 10000 = 95.97%


Testing the network...
Net_2 Accuracy: 98.90188103711235 %


Testing the network...
Net_3 Accuracy: 96.65480427046263 %


Testing the network...
Net_4 Accuracy: 96.66497203863752 %


Epsilon: 0.4	Test Accuracy = 9492 / 10000 = 94.92%


Testing the network...
Net_2 Accuracy: 98.65717192268566 %


Testing the network...
Net_3 Accuracy: 96.4089521871821 %


Testing the network...
Net_4 Accuracy: 96.26653102746694 %


Epsilon: 0.6	Test Accuracy = 8266 / 10000 = 82.66%


Testing the network...
Net_2 Accuracy: 94.18817313554155 %


Testing the network...
Net_3 Accuracy: 92.03413940256046 %


Testing the network...
Net_4 Accuracy: 89.53464742938428 %
```

```
Epsilon: 0.8      Test Accuracy = 4748 / 10000 = 47.48%


Testing the network...
Net_2 Accuracy: 66.05597964376591 %


Testing the network...
Net_3 Accuracy: 67.58269720101782 %


Testing the network...
Net_4 Accuracy: 56.51908396946565 %


Epsilon: 1.0      Test Accuracy = 1514 / 10000 = 15.14%


Testing the network...
Net_2 Accuracy: 29.148892951452364 %


Testing the network...
Net_3 Accuracy: 40.71704245378834 %


Testing the network...
Net_4 Accuracy: 32.805200081251265 %
```

The same instructions are performed in the following cells, where, respectively, Network A’, Network B and Network B’ are chosen as targets and then the remaining are tested against the adversarial examples.

An additional figure is reported showing the Accuracy of the target network as a function of Epsilon.

To conclude, for each value of epsilon a set of 5 elements is reported: each element is an adversarial example generated as a perturbed image of the original image.

ANALYSIS OF RESULTS – PART I OF THE ASSIGNMENT

In the first part of the assignment, the goal is to produce a table where accuracies for each combination of Target networks of adversarial attacks and Networks for evaluation of accuracy is reported. Notice that in this part is not required to try different set of epsilons, but the code as been implemented to have Part I and Part II together. For considerations in the Part I, I will refer to EPSILON = 0.6.

For this value, the following table has been obtained

TABLE OF RESULTS

Epsilon: 0.6	NET A	NET A’	NET B	NET B’
Target: NET A	82.66 %	94.19 %	92.03 %	89.53 %
Target: NET A’	93.47 %	78.21 %	91.47 %	88.87 %
Target: NET B	94.45 %	93.75 %	45.11 %	64.54 %
Target: NET B’	95.18 %	93.64 %	70.48 %	39.12 %

In red, elements along the diagonal of the matrix are highlighted. These elements correspond to the values of the accuracies of the Target Networks when tested against the generated adversarial examples. The other elements are the values of the accuracies of the other Networks when tested against generated adversarial samples for a different Target Network.

From these results, some considerations can be extracted:

1) Notice that in every row, the accuracy of the Target Network is always the worst. This is reasonable since FGSM is creating perturbed images with the goal of maximize the probability of fooling the target network. In a certain sense, adversarial examples are “optimized” to be the worst testing samples for Target Network, but not for the others.

2) With a value of 0.6, the original accuracy of the Target Networks is always deprecated but you can notice that the two networks with A architecture are less deprecated with respect to the two networks with B architecture. We can say that network A is “more difficult” to be fooled by FSGM because there is coherence between A and A’ accuracy (82 %, 78 %) and between B and B’ accuracy (45 %, 39 %), however A and A’ accuracies are way better with respect to B and B’ accuracies, that drastically drop.

3) Even if the target network is the one where accuracy is reduced the most, you can notice that adversarial examples for the target network can partially fool also the other networks! As explained in the Introduction section, this is the key concept that explains why adversarial examples research topic is fundamental.

4) You can notice that there are some correlations between the structure of the networks and they robustness with respect to adversarial examples. Consider for example when Target Network is Network A: you can see that accuracy of network A’ is also reduced by 5 %, which can be considered quite significant. This is an evidence that even if two networks share the same structure, the effect of the same adversarial examples on them is different! However, the amount of this effect changes: the effect is always the worst in the target network because perturbed images are computed from the target network gradient and gradient depends on the initial parameters values! If the initial weights are different, even the gradient will be different and so the effect.

5) Notice that accuracy is reduced even in network with different architectures. Considering always example of Target Network A: adversarial examples reduce in both case accuracies of networks B and B’ by more than 5 %, which is a relevant amount.

6) There is consistency in the result, in the sense that when A and A’ are the targets, the accuracies of B and B’ are both around 90 % (an average reduction of 7 % from original accuracy); on the other hand, when B and B’ are the targets, the accuracies of A and A’ are both around 94 % (an average reduction of 5 % from original accuracy). We can say then that A/A’ networks are more robust with respect to adversarial examples generated with B/B’ as targets, and B/B’ are less robust with respect to adversarial examples generated with A/A’ as targets.

7) Notice also that when B is the target, adversarial examples affects way more network B’ accuracy (64 %) with respect to A and A’ accuracy (94 % ca.). Same happens when B’ is the target: B accuracy also drops to (70 %) while A and A’ accuracies remain at 94 %.

ANALYSIS OF RESULTS – PART II OF THE ASSIGNMENT

In the part II of the assignment, the request was to try a different set of epsilons. As explained in the code explanation section, this has been implemented with a for loop where epsilons values are the following:

```
epsilons = [0.1, .2, .4, .6, .8, 1.0]
```

The observations for epsilon = 0.6 have been already given in the PART I explanation, but here are reported the other tables generated from the different values of epsilons:

Epsilon: 0.1	NET A	NET A’	NET B	NET B’
Target: NET A	96.66 %	99.14 %	96.77 %	96.78 %
Target: NET A’	98.97 %	97.51 %	96.75 %	96.78 %
Target: NET B	99.22 %	99.37 %	90.28 %	98.85 %
Target: NET B’	99.26 %	99.29 %	98.64 %	89.41 %

Epsilon: 0.2	NET A	NET A’	NET B	NET B’
Target: NET A	95.97 %	98.90 %	96.65 %	96.66 %
Target: NET A’	98.77 %	97.09 %	96.61 %	96.59 %
Target: NET B	99.03 %	99.22 %	89.06 %	98.43 %
Target: NET B’	98.97 %	99.16 %	98.28 %	88.09 %

Epsilon: 0.4	NET A	NET A’	NET B	NET B’
Target: NET A	94.92 %	98.66 %	96.41 %	96.27 %
Target: NET A’	98.23 %	96.00 %	96.03 %	96.13 %
Target: NET B	98.67 %	98.86 %	85.86 %	97.10 %
Target: NET B’	98.58 %	98.76%	96.81 %	84.78 %

Epsilon: 0.8	NET A	NET A’	NET B	NET B’
Target: NET A	47.48 %	66.06 %	67.58 %	56.51 %

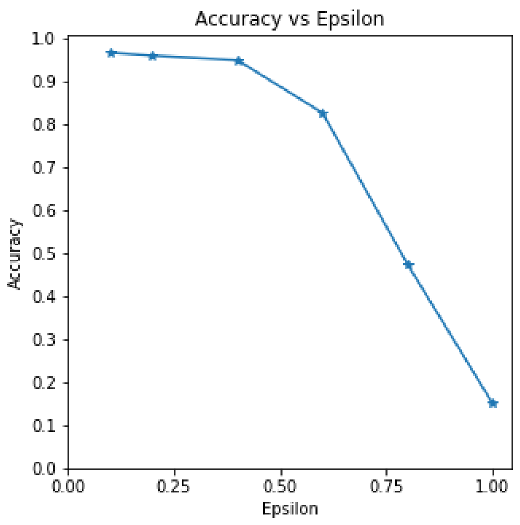
Target: NET A’	67.91 %	27.00 %	68.36 %	58.09 %
Target: NET B	62.13 %	59.57 %	12.42 %	18.55 %
Target: NET B’	62.79 %	55.40 %	21.32 %	9.96 %

Epsilon: 1.0	NET A	NET A’	NET B	NET B’
Target: NET A	15.14 %	29.14 %	40.72 %	32.81 %
Target: NET A’	29.64 %	9.05 %	40.11 %	32.91 %
Target: NET B	16.70 %	19.87 %	2.22 %	5.67 %
Target: NET B’	10.83 %	12.70 %	3.58 %	1.85 %

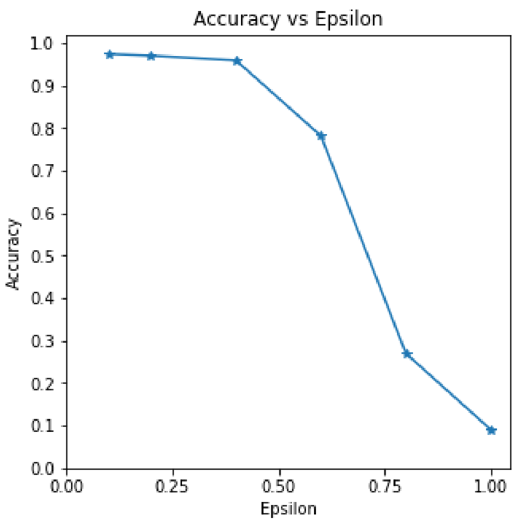
To analyse the effect of the change of epsilon value, also the following figures have been generated.

These figures report the accuracy vs epsilon value of the target networks:

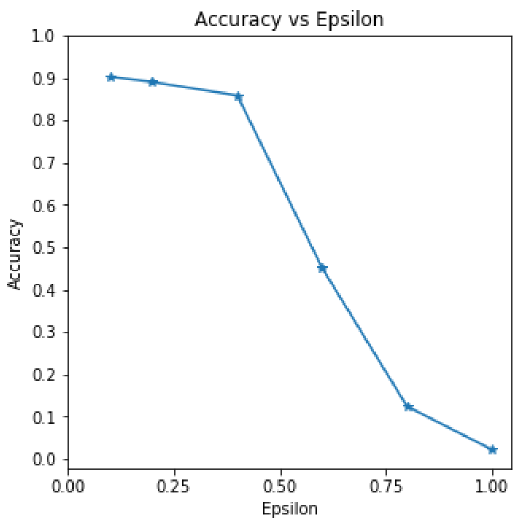
NETWORK A:

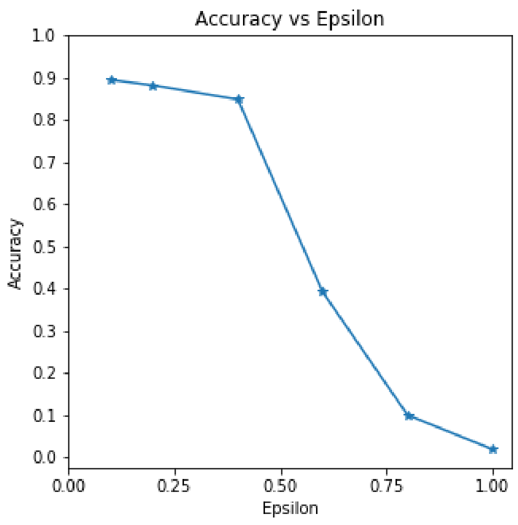


NETWORK A’:



NETWORK B: NETWORK B’:





From these figures and the resulting tables, we can extract some fundamental considerations:

1) It is evident, and also reasonable, that the bigger the value of epsilons, the lower will be the accuracy of the Target Network when tested against the generated adversarial examples. This is because epsilon represents the amount of superimposed noise to the original image: the bigger the noise, the more difficult is for the network to classify correctly the input.

2) We have already discussed how networks with the same architecture but different initial weights can lead to different results, and this is confirmed by the fact that plot of accuracy vs epsilon of A is different from the one of A’ (and same for B and B’). Notice however that the couple (A,A’) and (B,B’) share a “pattern”. In fact, you can see that when epsilon increase, the accuracy decrease, but there is an epsilon value when accuracy of the network abruptly changes. For network A and A’, this value is after Epsilon = 0.5, while for B and B’ this value is after Epsilon = 0.35! This partially confirms what already stated above, that is that network with A architecture are a bit more “difficult” to fool with respect to network B, and for same accuracy value the adversarial examples have more noise.

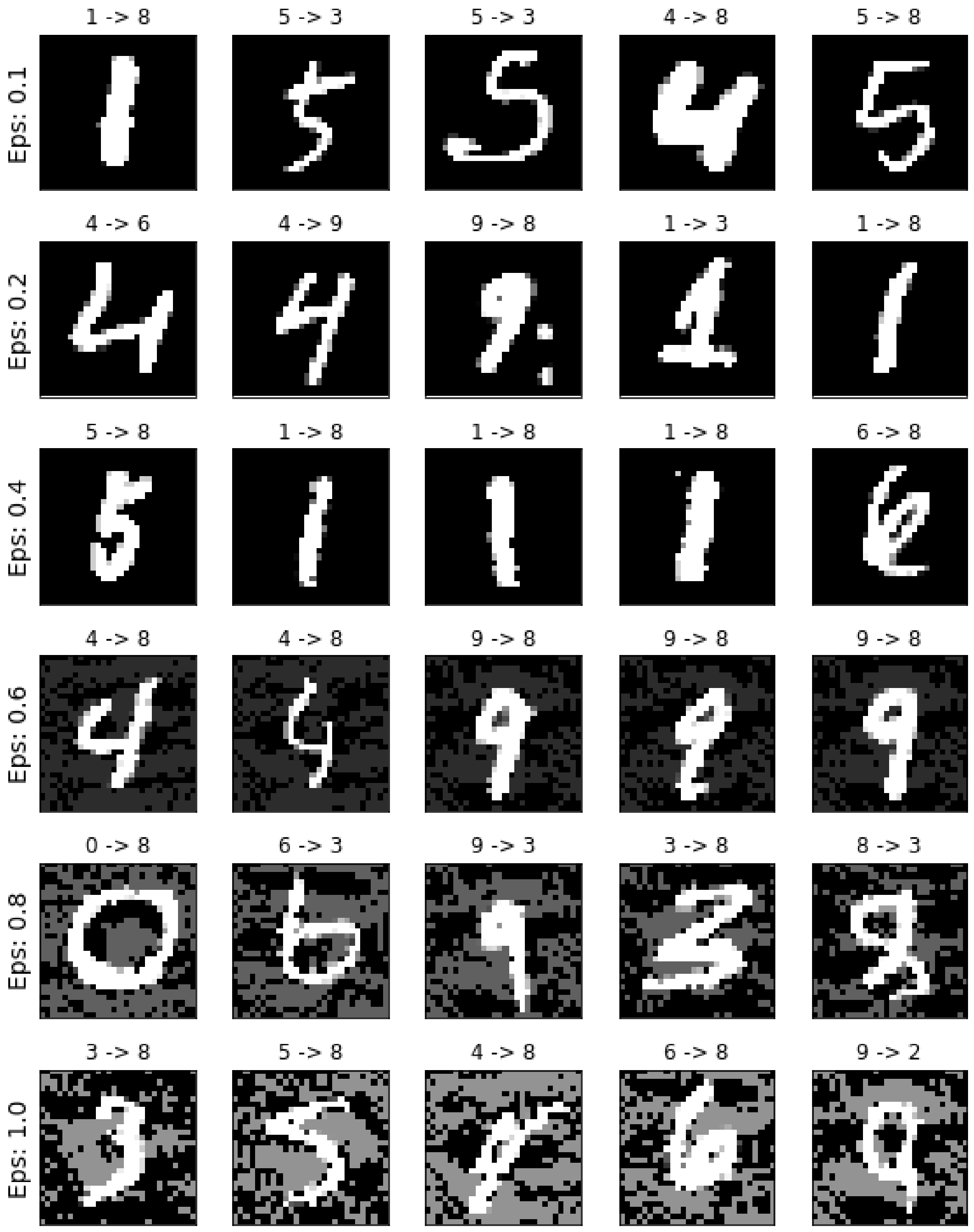
3) For the same value of epsilon, accuracy of Network A, A’ as Target is higher with respect to accuracy of Network B, B’ as Target. This means that networks with architecture of type A are more robust with respect to type B.

4) For values of epsilon above 0.5, consideration stated above for the epsilon = 0.6 case are even more evident. Take into considerations, for example, epsilon = 0.8. When Network A is the target its accuracy is of 47 % while network A’ accuracy is still at 66 %. This drop is of course significant with respect to original accuracy, but it is still much higher than target network accuracy: it is another confirmation that even if two networks share the structure, different initialization can lead to very different results.

5) Above epsilon = 0.5, we said that B and B’ accuracy are badly deprecated. In particular, when B is the target also B’ accuracy drops and viceversa. However, in this cases accuracies of networks A and A’ decreases but remain above 60%. It is another proof that support the robustness of net A and A’ with respect to adversarial examples on target net B and B’.

To conclude this analysis, I want to report an example of the generated adversarial examples for the different values of epsilon: you can notice that noticeable changes to human eye are for values of epsilon greater than 0.4. For this value, accuracy of the networks is already deprecated (in particular, accuracy of B and B’ is reduced of 10%). Even for value of epsilon = 1, where the precision of the networks drastically drops to 1-5 %, we are still able to easily classify the images.

I wanted to highlight this fact because it is still a topic of research why neural networks are so badly affected by this changes that are imperceptible for us.


Eps: 0.1
1 -> 8
5 -> 3
5 -> 3
4 -> 8
5 -> 8
Eps: 0.2
4 -> 6
4 -> 9
9 -> 8
1 -> 3
1 -> 8
Eps: 0.4
5 -> 8
1 -> 8
1 -> 8
1 -> 8
6 -> 8
Eps: 0.6
4 -> 8
4 -> 8
9 -> 8
9 -> 8
9 -> 8
Eps: 0.8
0 -> 8
6 -> 3
9 -> 3
3 -> 8
8 -> 3
Eps: 1.0
3 -> 8
5 -> 8
4 -> 8
6 -> 8
9 -> 2