Adversarial Robustness

Explored by Geetika Vadali

Core question to adversarial robustness is can we develop classifiers that are robust to test time perturbations of their inputs? These perturbations are caused by an adversary intent on fooling the classifier to misclassify.

From the concept of gradient descent where the gradient computes how a small adjustment to the parameters of model affect the loss function. This gradient of loss can also be calculated with respect to the input x itself -> this will tell how a small change in x (our input) affects the loss function.

Creating an adversarial example

We want to adjust (perturb) input x so as to maximise loss. But this perturbation has to done by ensuring that by any “humanly reasonable definition”, the actual semantic of the content doesn’t change. But the classifier’s confidence plummets.

$$\underset{\delta \in \Delta}{\text{maximise}}\ \ell\left(h_\theta(x+\delta), y\right)$$

Where ∆ denotes the allowable set of perturbations. A common perturbation set to use is

ℓ_∞ ball

$$\Delta : \{\delta : \|\delta\|_\infty \leq \varepsilon\}$$

where $\|z\|_\infty = \max_i |z_i|$

Targeted attack - where we maximise the loss of the correct class and simultaneously minimize the loss of target class.

$$\max_{\delta \in \Delta} \left[\ell\left(h_\theta(x+\delta), y\right) - \ell\left(h_\theta(x+\delta), y_{target}\right) \right]$$

Traditional Training

Traditional Risk

Risk

$$R(h_\theta) = \mathbb{E}_{(x,y)\sim D}\left[\ell(h_\theta(x)), y)\right]$$

Empirical Risk

$$\hat{R}(h_\theta, D) = \frac{1}{|D|}\sum_{(x,y)\in D} \ell(h_\theta(x), y)$$

Traditional process of training an ML algorithm = minimizing empirical risk on some training data set D_{train}

$$\min_\theta \hat{R}(h_\theta, D_{train})$$

Adversarial Training

Instead of suffering the loss of each sample point from training data, only the worst case loss in some region around the sample point is suffered. Thus, the risk here measures the worst case loss of the classifier, if we are able to adversarially manipulate every input in the dataset within its allowable set of Δ.

Adversarial Risk

$$R_{adv}(h_\theta) = \mathbb{E}_{(x,y)\sim D}\left[\max_{\delta \in \Delta(x)} \ell(h_\theta(x+\delta), y)\right]$$

empirical adversarial risk

$$\hat{R}_{adv}(h_\theta, D) = \frac{1}{|D|} \sum_{(x,y)\in D} \max_{\delta \in \Delta(x)} \ell(h_\theta(x+\delta), y)$$

When the hypothesis class is linear

$$\min_{W,b} \frac{1}{|D|} \sum_{x,y \in D} \max_{\|\delta\| \leq \epsilon} \ell\left(W(x+\delta) + b, y\right)$$

Through this formulation, we can solve the inner maximization for binary optimization and provide a relatively tight upper bound for the case of multi-class classification. This will also result in a globally optimal robust classifier.
This is not the case with the deep network case where neither of the optimizations can be done globally due to the non-convexity of the network itself.

Binary Classification

$$h_\theta(x) = w^T x + b$$

$$\text{for } \theta = \{ w \in \mathbb{R}^n, b \in \mathbb{R} \}$$

$$y = \{+1, -1\}$$

$$l(h_\theta(x), y) = \log\left(1 + \exp(-y \cdot h_\theta(x))\right) \equiv \mathcal{L}(y \cdot h_\theta(x))$$

Hence for binary classification, the probability of predicting +1 for a data point x would look like -

$$p(y = +1 \mid x) = \frac{1}{1 + \exp(-h_\theta(x))}$$

Optimal perturbation

$$\delta^* = -y\,\varepsilon \cdot \mathrm{sign}(w)$$

Optimal perturbation is not dependent on x (training example), that implies that the best perturbation that can be applied is common for all training examples.

In a standard linear model, we can perform exact robust optimization by incorporating the l1 norm and subtracting term of following form (this for the special case of binary classification where labels are -1/+1)

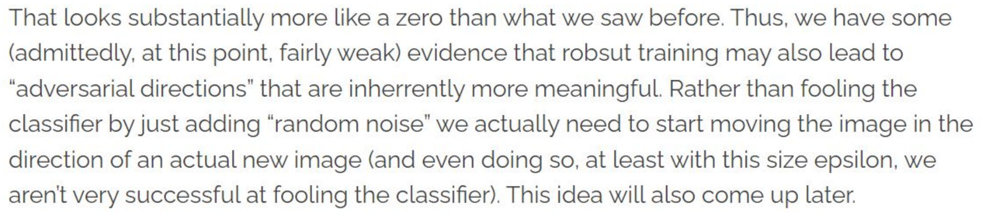
$$\varepsilon(2y-1)\,\|w\|_1$$

Robustness

Robust optimization is worst-case adversarial optimization.

There is a fundamental trade-off between clean accuracy and robust accuracy and as much as we can get better at robustness, we compromise on the clean test error.

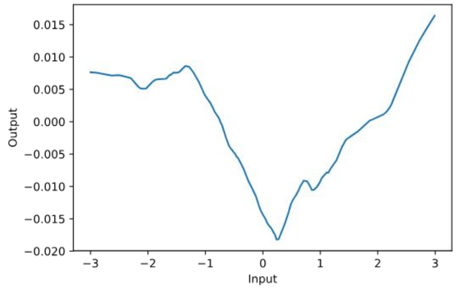
Robust training leads to ‘adversarial directions’??



That looks substantially more like a zero than what we saw before. Thus, we have some (admittedly, at this point, fairly weak) evidence that robsut training may also lead to “adversarial directions” that are inherrently more meaningful. Rather than fooling the classifier by just adding “random noise” we actually need to start moving the image in the direction of an actual new image (and even doing so, at least with this size epsilon, we aren't very successful at fooling the classifier). This idea will also come up later.

Neural Networks

Problems with finding optimal adversarial example for NNs:



- Loss surface wrt the input (not parameter) is irregular and bumpy. This causes two problems:
- In high dimensions, the loss is highly sensitive to even very small perturbations due to steep gradients at many points along surface.
- It is not easy to solve the inner maximisation problem over the perturbation because the loss surface is not purely convex. Will suffer from local optimas - cannot solve the true robust optimization problem to find the optimal perturbation.
- Approximately solving the inner maximisation problem then : 1. Lower bound, 2. Exact optimization (can be done in some by combinatorial methods) and 3. Upper bounds on relaxations

Taxonomy of adversarial attacks on the basis of threat model

1. White box attacks - attacker has access to the model’s parameters
2. Black box attacks - attacker does not have access to model’s parameters and hence uses a different model or no model to generate adversarial examples.

White Box attacks -

- Fast Gradient Sign Method (FGSM) : computes an adversarial image by adding a pixel-wise perturbation of magnitude in the direction of the gradient. *(I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” arXiv preprint arXiv:1412.6572, 2014.)*

- Projected gradient descent

  Iterate over gradient ascents on smaller step size and doesn’t consider constraints on amount of time and efforts. Proj{.} will project the updated adversarial sample into the epsilon neighbourhood and a valid range.

- DeepFool [1]
- Carlini and Wagner attack [2]
- Jacobian based saliency map attack [3]
- Universal adversarial attack [4]

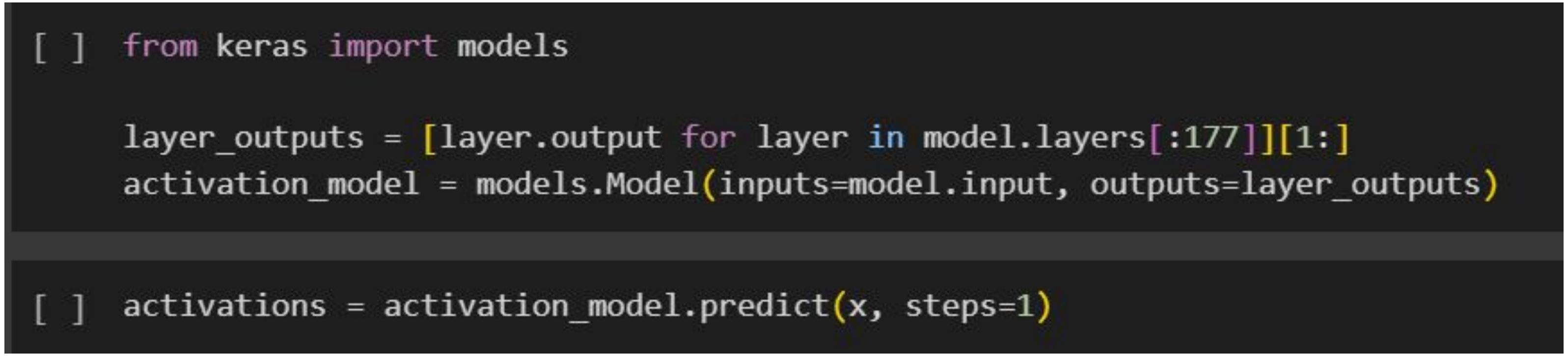
[1] : DeepFool: a simple and accurate method to fool deep neural networks, Moosavi-Dezfooli et al., 2016

[2] : Towards Evaluating the Robustness of Neural Networks, Carlini and Wagner, 2016

[3] : Maximal Jacobian-based Saliency Map Attack, Xu and Wiyatno, 2018

[4] : Universal adversarial perturbations, Moosavi-Dezfooli et al., 2017

Adversarial_1 code results

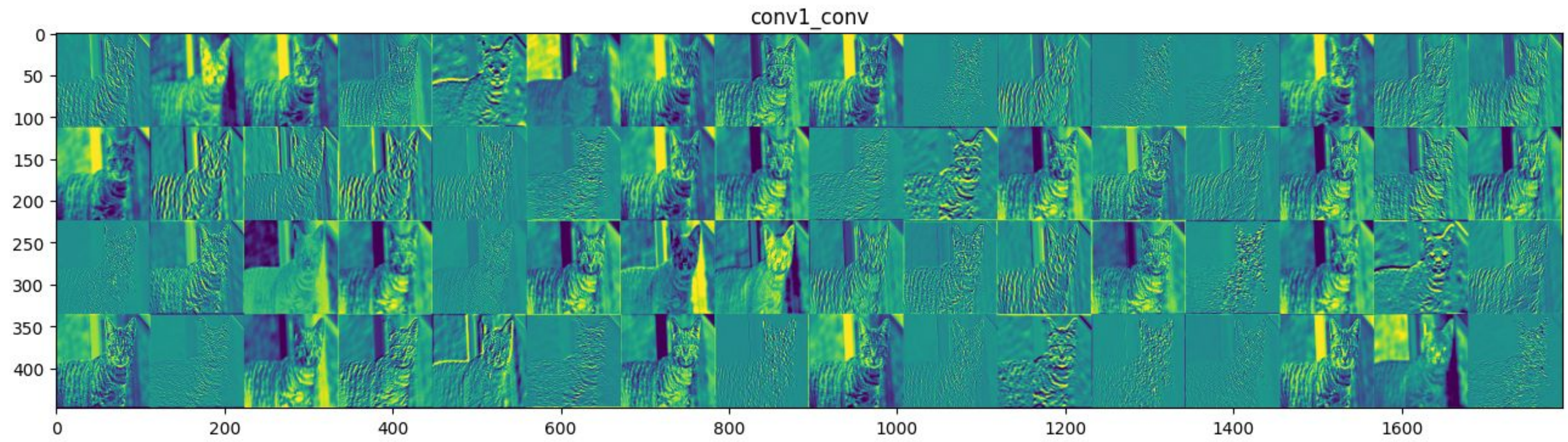


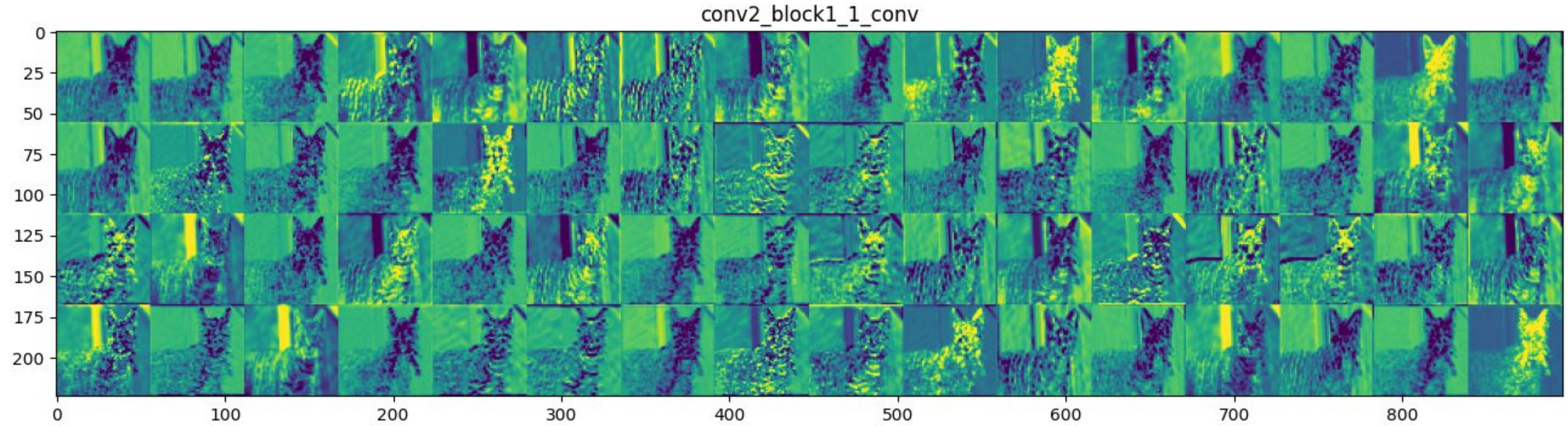
```
[ ]  from keras import models

     layer_outputs = [layer.output for layer in model.layers[:177]][1:]
     activation_model = models.Model(inputs=model.input, outputs=layer_outputs)
```

```
[ ]  activations = activation_model.predict(x, steps=1)
```

`activations` represents a numpy array containing the intermediate layer activations of all the 177 layers for given input image tensor x.

`activations[50][0][0][0]` is trying to access the value at the first position (index 0) along all dimensions of the 51st layer's activations tensor.

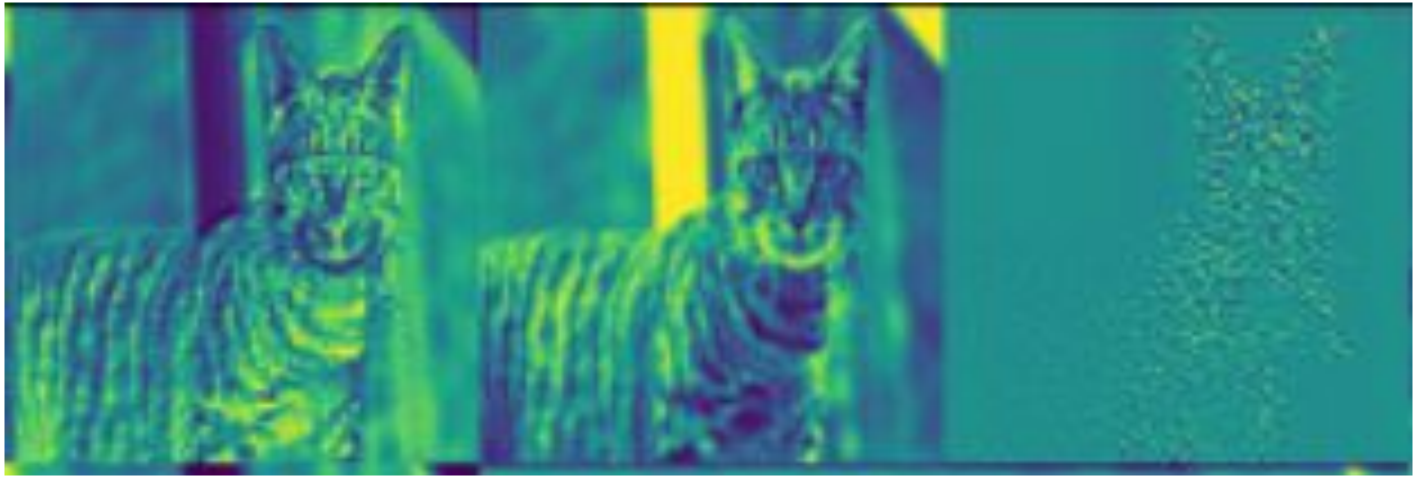

conv1_conv
0
50
100
150
200
250
300
350
400
0
200
400
600
800
1000
1200
1400
1600


conv2_block1_1_conv
0
25
50
75
100
125
150
175
200
0
100
200
300
400
500
600
700
800

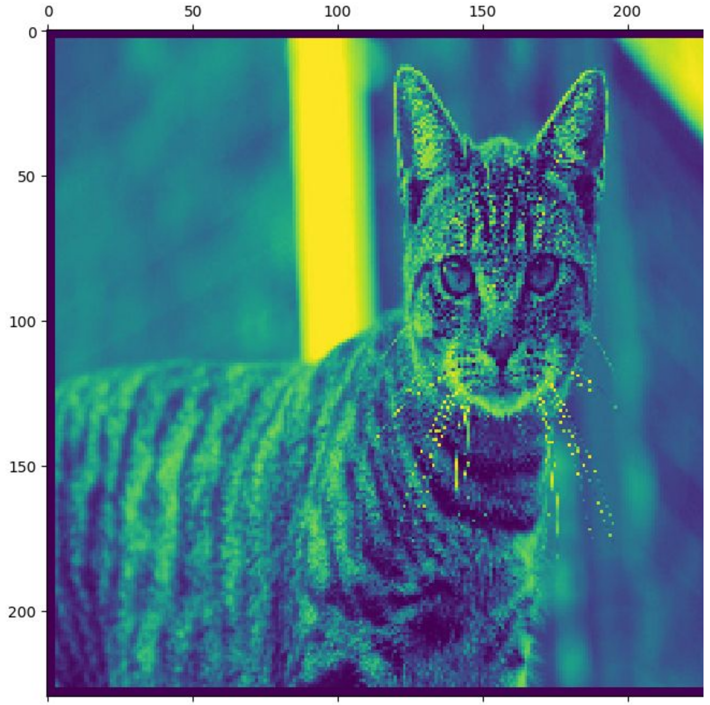
High negative activation

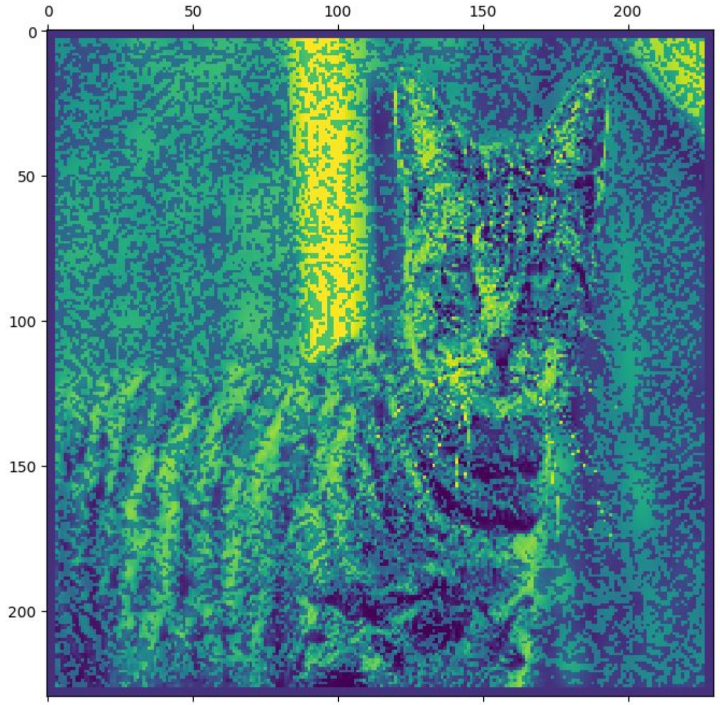
High positive activation

Low activation

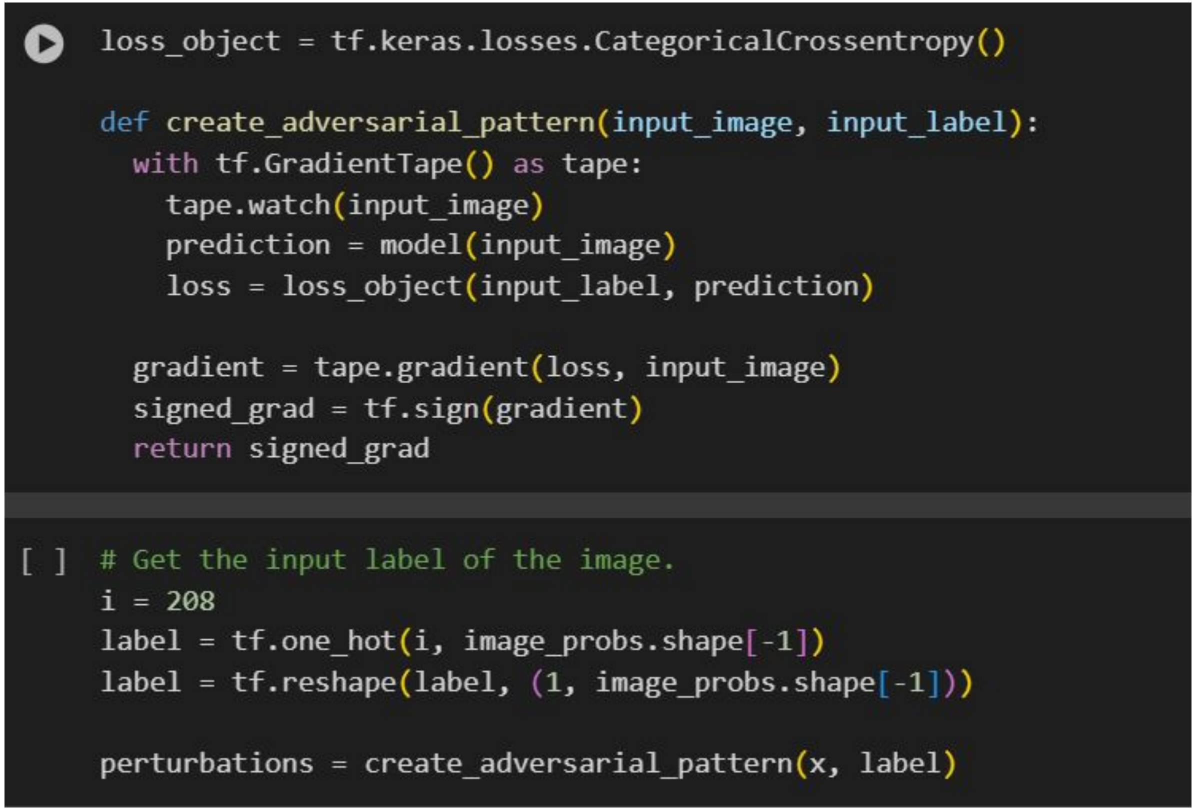


Adversarial attack - before and after FGSM



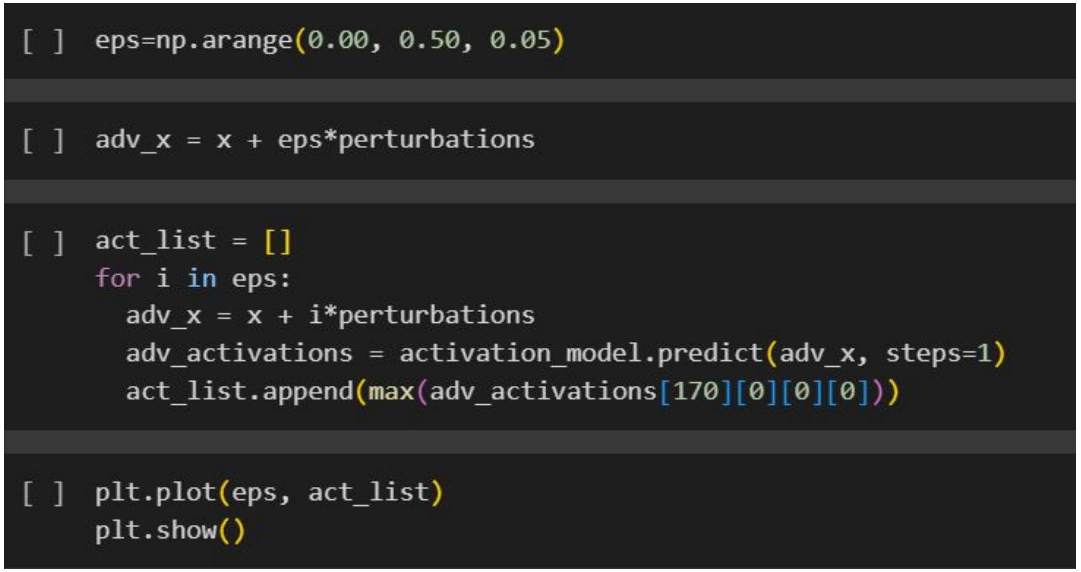


Fast Gradient Sign Method Adversarial Attack



```
loss_object = tf.keras.losses.CategoricalCrossentropy()

def create_adversarial_pattern(input_image, input_label):
  with tf.GradientTape() as tape:
    tape.watch(input_image)
    prediction = model(input_image)
    loss = loss_object(input_label, prediction)

  gradient = tape.gradient(loss, input_image)
  signed_grad = tf.sign(gradient)
  return signed_grad
```

```
[ ]  # Get the input label of the image.
     i = 208
     label = tf.one_hot(i, image_probs.shape[-1])
     label = tf.reshape(label, (1, image_probs.shape[-1]))

     perturbations = create_adversarial_pattern(x, label)
```



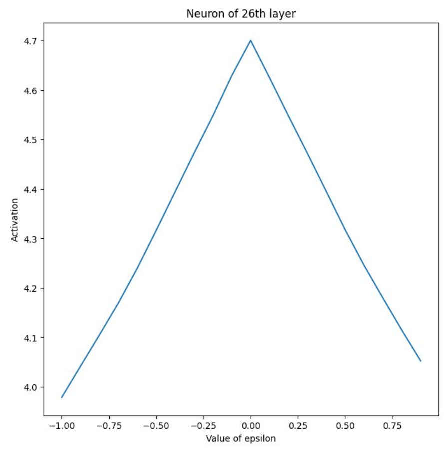
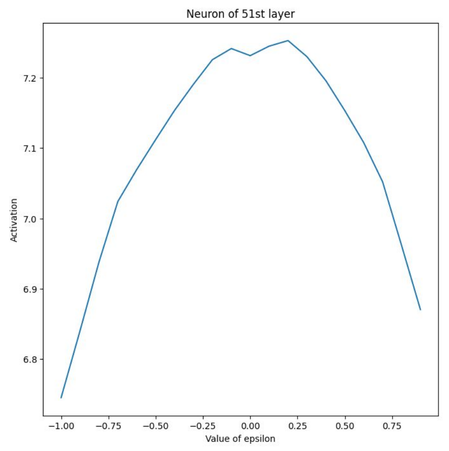
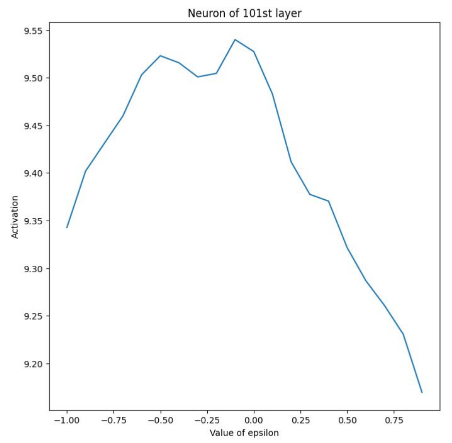
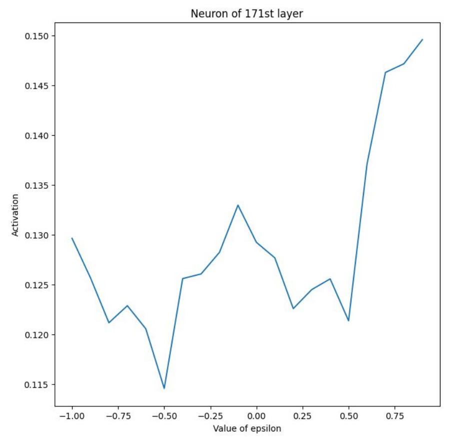
```
[ ]  eps=np.arange(0.00, 0.50, 0.05)
```

```
[ ]  adv_x = x + eps*perturbations
```

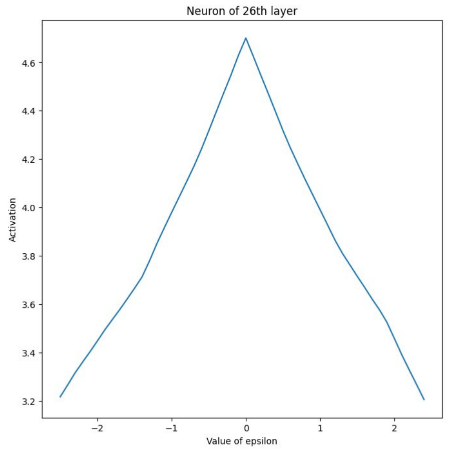
```
[ ]  act_list = []
     for i in eps:
       adv_x = x + i*perturbations
       adv_activations = activation_model.predict(adv_x, steps=1)
       act_list.append(max(adv_activations[170][0][0][0]))
```

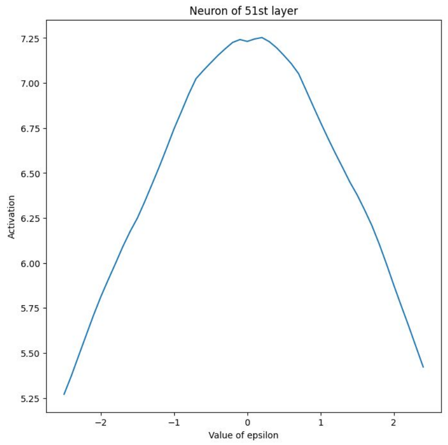
```
[ ]  plt.plot(eps, act_list)
     plt.show()
```

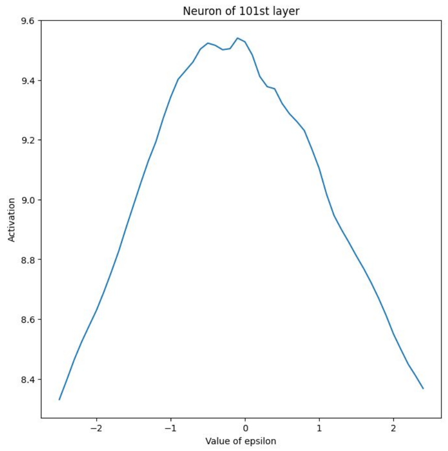
Defining a basic FGSM adversarial attack on the image and choosing maximum activation specifically for the nth layer (here 171st) of ResNet50. As epsilon (degree of perturbation) ranges from 0.00 to 0.50, we save this particular unit's activation for our image. Then visualise how activation changes with eps. And this is recorded for a number of layers

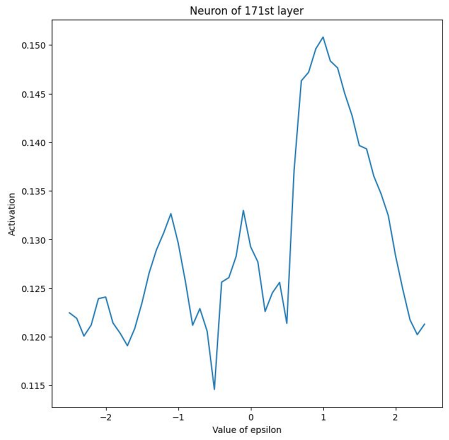
			
26th layer first unit Steepest (approximately probable linear) decrease in activations at positive degree of perturbation	51st Layer First Unit Activations increase and then keep decreasing (beyond eps=0.50 too)	101st Layer First Unit Observed decrease in activations but still not halved/near-zero	171st Layer First Unit Haphazard activation changes

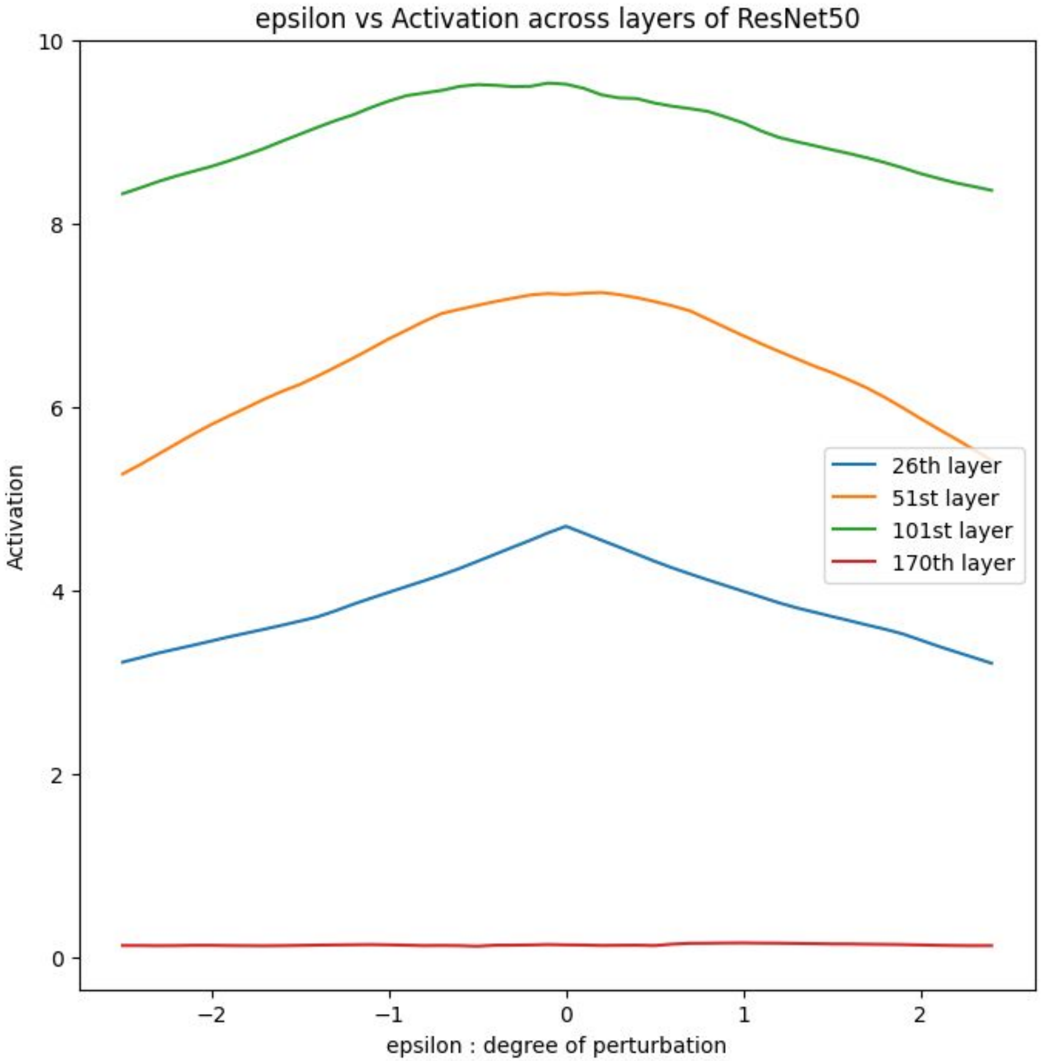
-2 to +2 epsilon for FGSM










epsilon vs Activation across layers of ResNet50
Activation
epsilon : degree of perturbation
0
2
4
6
8
10
−2
−1
0
1
2
26th layer
51st layer
101st layer
170th layer

From this, I think the sensitivity of perturbation to the activation of neurons in a layer decreases as we move deeper into the CNN model. More significant changes are when activations are less abstract and more knowledge of the image is retained. But simultaneously, the reactions of activations of deeper layers are extremely uncertain unless I am missing a logical reason behind it.

From the perspective of a normalised scale (range of values) - the drops in activation are maximum when the neuron layers come earlier. Which proves the intuition that earlier layers are more sensitive to adversarial perturbations.

Loss needs to be tweaked, we want to move in direction of reducing activation and not in direction of reducing cross entropy loss

Original Image Activation (x):

- You have an original image, let's call its activation value x. This activation value represents how the image affects a neural network or some other model.

Adversarial Image Activation (adv_x):

- You want to create an adversarial image with a specific activation value, denoted as adv_x.
- The goal is to make adv_x equal to R times the activation value of the original image x.

Epsilon (ε):

- Epsilon is the degree of perturbation you introduce to the original image x to create the adversarial image adv_x.
- Essentially, you're adding or subtracting a small amount to each pixel of the original image to change its activation value.

So, the task is to find the right degree of perturbation (epsilon) to apply to the original image x so that its activation value becomes R times original activation.

Knowns

Original image - **x**

Unit we are focussing on - **unit**

Original image activation - **x_act**

Adversarial image activation - **adv_x_act**

Target ratio to reduce orig activation by - **target_ratio**

Target activation - **target_act = target_ratio * x_act**

Goal is : find epsilon where adv_x_act == target_act

Where epsilon is used

adv_x = x + epsilon*perturbations

Where the adversarial image is formed by concatenating perturbation (noise) times a certain degree of perturbation (epsilon).

Perturbation is the signed gradient of the loss function (in our case the difference/change in the activation of adversarial example compared to original image of previous iteration) with respect to the original image.

Proposed algorithm

Depending on the convergence of adversarial activation towards the target activation, and the difference between those values

```
change = abs(adv_x_act_val - target_act)
eps += change*eps
adv_x = x + eps*perturbation
```

As epsilon is updated based on the change, this epsilon is used to generate a new adversarial example. Then the activation of this adversarial example is again iteratively compared to the target activation.

And as this differences decreases, activation is made to converge to target.

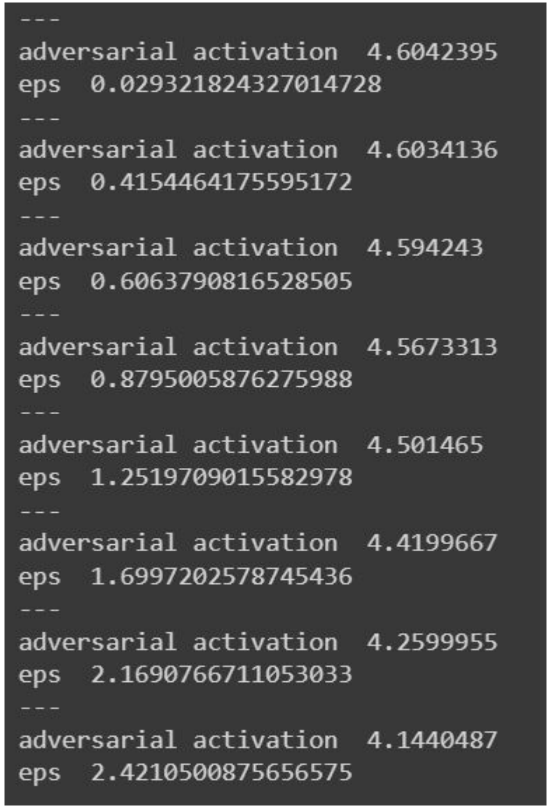
`control_activation(x, 0.9, 25, label)`

For the 25th unit of resnet50, reduce activation by 90%

Original x activation = 4.6042547

Target activation (90%) = 4.143829250335694

Initialised adversarial activation = 4.6042547



```
---
adversarial activation  4.6042395
eps  0.029321824327014728
---
adversarial activation  4.6034136
eps  0.4154464175595172
---
adversarial activation  4.594243
eps  0.6063790816528505
---
adversarial activation  4.5673313
eps  0.8795005876275988
---
adversarial activation  4.501465
eps  1.2519709015582978
---
adversarial activation  4.4199667
eps  1.6997202578745436
---
adversarial activation  4.2599955
eps  2.1690766711053033
---
adversarial activation  4.1440487
eps  2.4210500875656575
```

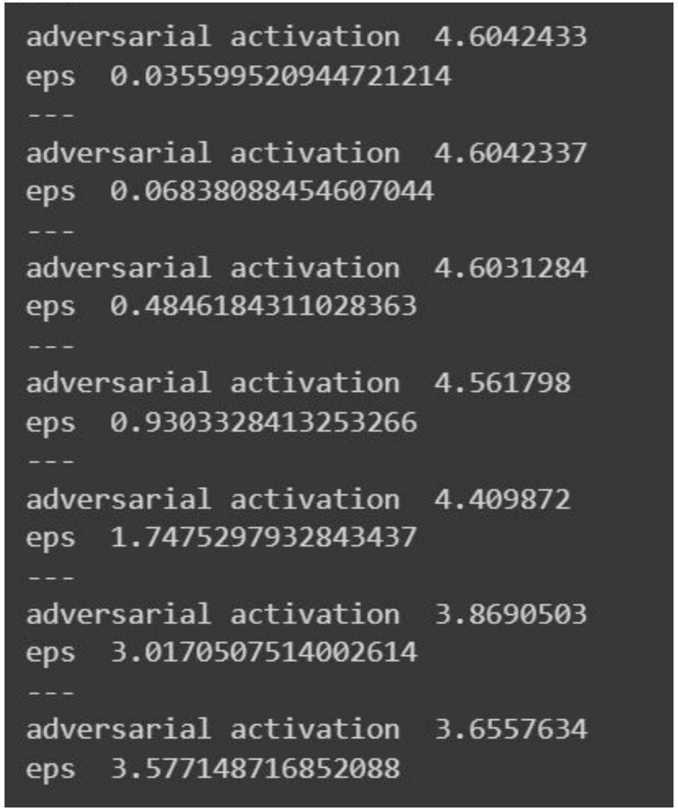
`control_activation(x, 0.8, 25, label)`

For the 25th unit of resnet50, reduce activation by 80%

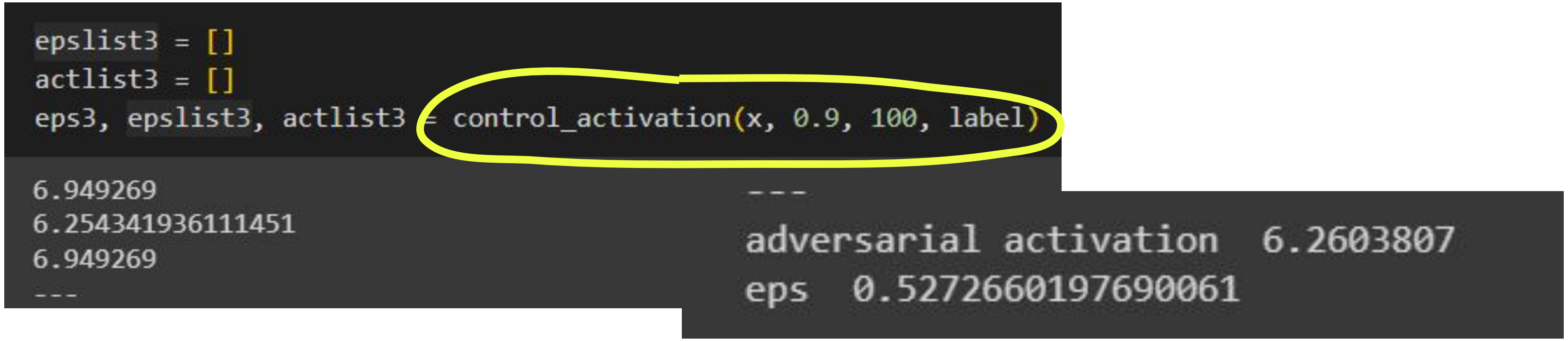
Original x activation = 4.6042547

Target activation (90%) = 3.6834060

Initialised adversarial activation = 4.6042547



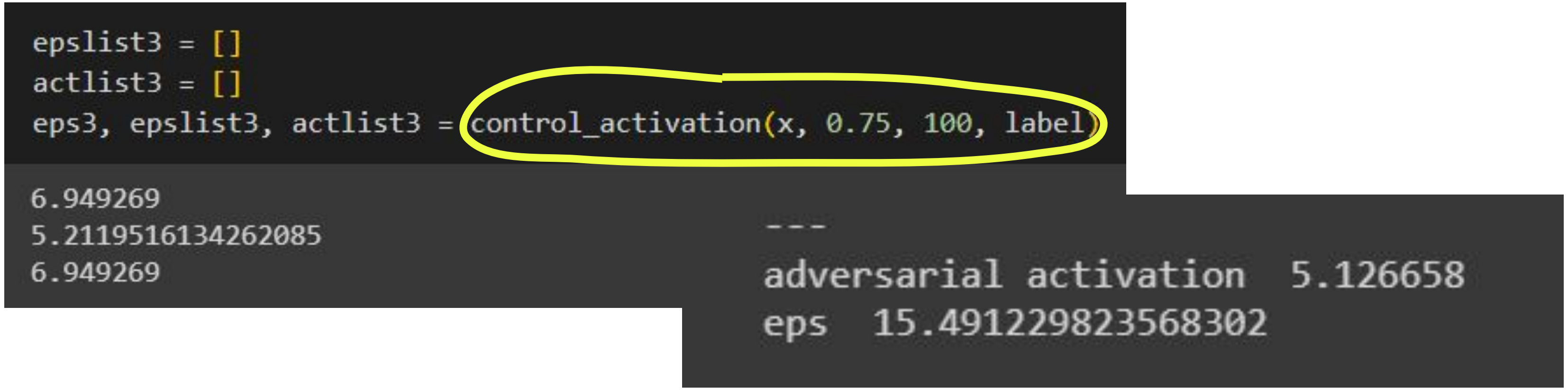
```
adversarial activation  4.6042433
eps  0.035599520944721214
---
adversarial activation  4.6042337
eps  0.06838088454607044
---
adversarial activation  4.6031284
eps  0.4846184311028363
---
adversarial activation  4.561798
eps  0.9303328413253266
---
adversarial activation  4.409872
eps  1.7475297932843437
---
adversarial activation  3.8690503
eps  3.0170507514002614
---
adversarial activation  3.6557634
eps  3.577148716852088
```



```
epslist3 = []
actlist3 = []
eps3, epslist3, actlist3 = control_activation(x, 0.9, 100, label)
```

```
6.949269
6.254341936111451
6.949269
---
```

```
---
adversarial activation  6.2603807
eps  0.5272660197690061
```



```
epslist3 = []
actlist3 = []
eps3, epslist3, actlist3 = control_activation(x, 0.75, 100, label)
```

```
6.949269
5.211951613426208 5
6.949269
```

```
---
adversarial activation  5.126658
eps  15.491229823568302
```