Implementing a KNN based TTE attack detector

Tanmay Ambadkar
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, 16803
ambadkar@psu.edu

Abstract

This report presents an implementation of a K*NN detector to detect TTE attacks. The detector first clusters the activations of the input data for every class present in the dataset and uses a KNN classifier on the cluster means obtained. This implementation yields a robust detector and can detect zeroth-order optimization (ZOO) attacked images with reasonable accuracy. Multiple experimental settings have been tested to select the best hyper-parameters and a decision statistic for the detector.

1 Introduction

The following assignment aims to implement a K*NN-based TTE attack detector. The model being used is the ResNet with four convolutional blocks and a linear layer for classification. The dataset used is the MNIST dataset, and the attack method is the ZOO attack. The report first explains the intuition behind using a K*NN detector. I explain the formulation of the detector next and the experimental setting to select the best hyper-parameters, which validate the intuition. The section after will explain the detection statistic, and the final section will show the results obtain.

The ZOO attack implementation and model used are from the last assignment.

2 Patch-wise ZOO-attack

In the last report, the patch-wise implementation was not clearly defined. In the proposed ZOO attack, I select a patch (3x3 or 4x4 block) and then optimise the image for that. Essentially, the coordinate descent is expanded to become a patch. What this does is increase the number of updates made to an image in a single optimization step, reducing the total steps required to attack an image. Another option would be to randomly select 9 coordinates in the image and optimize, which might yield the same results, but I have not experimented with the method. My attack methodology is illustrated in Figure 1

3 Intuition

When we train a deep learning model, the activations highlight the most important parts of the image, which can generate a good classification result. Every image belonging to a class will have similar characteristics, which can be identified by looking at the activations of the neural network. A nice visualization of activations has been provided by [1] (Figure 2)

At the shallower levels, we can make out the most important parts of the digit 2. The network captures the edges which define the shape of 2. As we go deeper, the representations are more abstract and cannot be understood by humans. These activation maps give us a hint on how to define our TTE detector.

Preprint. Under review.

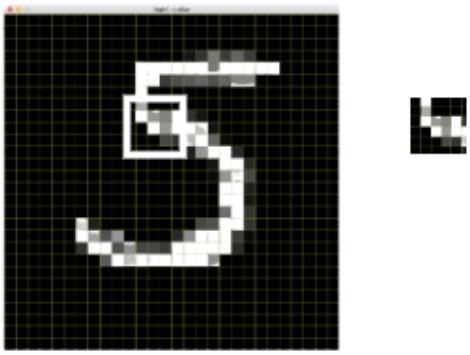


Figure 1: ZOO attack using Patch-wise gradient descent

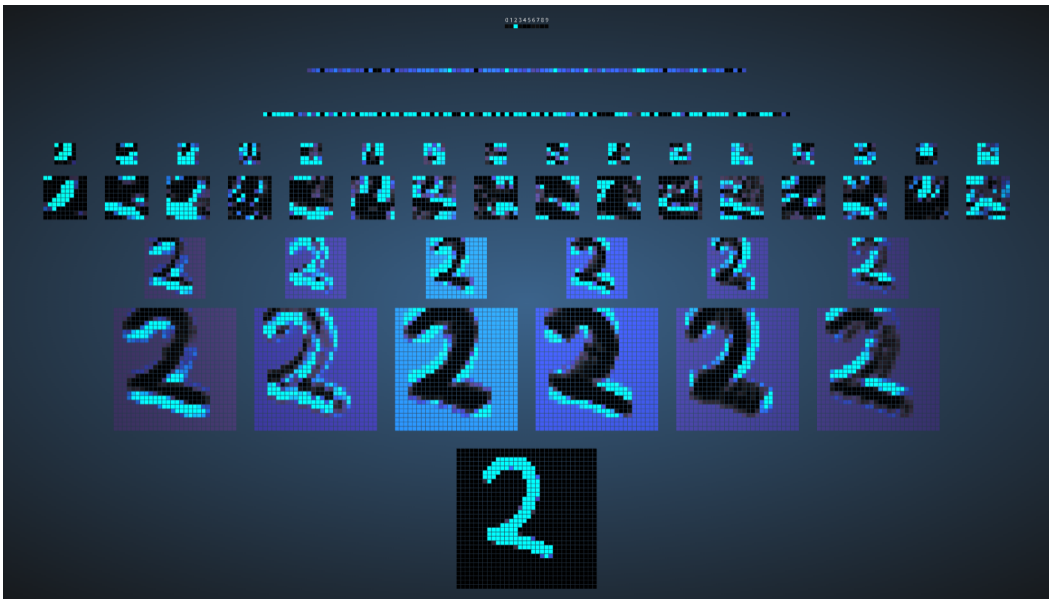


Figure 2: Activations of neural network for the MNIST digit 2

We can see that the activation maps for different digits will differ, so we can extract these activations and cluster them to get k different cluster centers for a class label $i \in 0, 9$. Then, any new activation will fall within one of these cluster centers. We can then use a distance-based voting rule to identify the class of any new image. This is the nearest neighbor rule popularly used in classification.

Upon inspection of the activations at different levels, we can infer that the activations at the shallower levels will have a lower distance in the cluster centers for different classes than at the deeper levels. This is because the maps differ less in the shallower layers than in the deeper ones, because the representations captured at deeper levels are more specific towards a digit, thus significantly different from each other. When a new point is to be classified, the distance between the cluster centers for the true class will be significantly smaller than the wrong class cluster centers for shallower layers than in deeper layers. Thus, it makes sense to use shallower layers to build the TTE detector than the deeper layers. Experimental results validate our hypothesis.

If we get an attacked image, the initial activations will still look like a non-attacked image. What happens at the deeper layers due to the attacked points causes a misclassification. Thus, when we use the nearest neighbor rule, the first neighbor will always return the correct class, while the next nearest neighbor might correspond to the attacked class.

In addition, we test the performance of our KNN classifier versus the DNN classifier on clean test sets to check the performance. Here, we will see that deeper layer activations give better performance. This is because at deeper layers, the embedded representations differ significantly to give a higher confidence, which in turn help the distance metric nearest neighbors classifier.

Thus, our intuition works in two ways. Deeper layers give better classification accuracy on clean sets than shallower layers by using fine-grained details, while shallower layers are more robust on attacked images by looking at coarse details.

I propose a detection statistic in the methodology section that uses the K*NN but discards the voting role, rather uses the distance and K neighbors to detect attacks with a better accuracy.

4 Methodology and TTE Detection statistic

Using the developed intuition, we first extract the activation maps for all convolutional blocks in the neural network. We have four such blocks in the network. We choose K clusters for every class i in the dataset and use the K-means algorithm to obtain cluster centers for the activations corresponding to the class i. Thus, we have a set of K * N data points, where N is the number of classes. Using these data points, we create a nearest-neighbors classifier, which can classify non-attacked and attacked images with good accuracy, and thus becomes a robust classifier.

However, we can make the attack detector even better. For an image, we get K neighbors along with their distances. If the predicted label is present in one of the K neighbors, we find the one with the least distance and compare it with a threshold. If the distance is less than the threshold, the image is labeled as not attacked. If it is greater, it means that the image is attacked. This will mitigate false positives, where a digit might look like some other but not. For example, 6 and 0 can easily be misclassified.

To decide the threshold, we vary it from 0 to 100, plot a ROC curve, and measure the area under it. We find the point in the ROC curve closest to (0, 1) and use that as our decision threshold.

This goes against our intuition that shallower layers will perform better. In this method, the decision threshold and the distances calculated matter greatly, along with the number of neighbors being considered. The number of neighbors determines what points are to be considered. It is possible that the attack yields top 4 neighbors as the attacked class and the 5th neighbor as the true class. The KNN rule will fail here, giving us a false negative. Our rule will ensure that the 5th neighbor is considered and checked within a threshold. If the threshold check fails, then we can consider it as not attacked. If it falls within the threshold, then the image is attacked. Thus, the predicted label is required here.

The deeper layers will have clusters much farther from each other. Also, the ZOO attack implemented tries to misclassify by a very small margin. Thus, the activations in the initial layers will have all distances within a threshold and perform poorly. In the deeper layers, the distances will be greater. Thus, the threshold will greatly affect performance. Experimental results show that any layer will perform well, but the difference is not as large.

The steps to create the TTE detector are:

1. Use a validation set to get activations and their corresponding labels.
2. Perform K-means clustering on activations for a label i, and store cluster centers. Repeat this step for all classes in the dataset.
3. Using cluster centers and their corresponding labels, build a KNN classifier with k neighbors using the euclidean distance.
4. Test this classifier's performance on a test set and compare with DNN accuracy. Tune hyperparameters of K-means and KNN if necessary
5. Select a distance threshold between 0 and 100. This will be the TTE detection statistic.
6. For an image, get corresponding activations and pass through KNN to get top K neighbors. If the predicted DNN label is part of the top K neighbors, get the first occurring index and its distance.
7. If the distance is less than threshold, the image is not attacked. If > than threshold, the image is attacked.

8. Vary this decision threshold and plot a ROC curve. The point where the (fpr, tpr) $\approx$ (0, 1) is the best threshold.

Thus, using this method, we have a robust classifier and an attack detector.

4.1 Why does this threshold method work?

One might question the logic or merit of using such a detection method. If the clusters are well-defined and separated by some hyperplane for each class, this method is not required. The K*NN rule will always work. Which is correct, except it is quite difficult to visualize embeddings in dimensions > 3 to prove the concept. We are creating clusters for 1 class rather than all classes; thus, cluster centers for 2 different classes will likely be very close. For the sake of simplicity, I will represent embeddings in 2 dimensions for 2 classes (Figure 3).

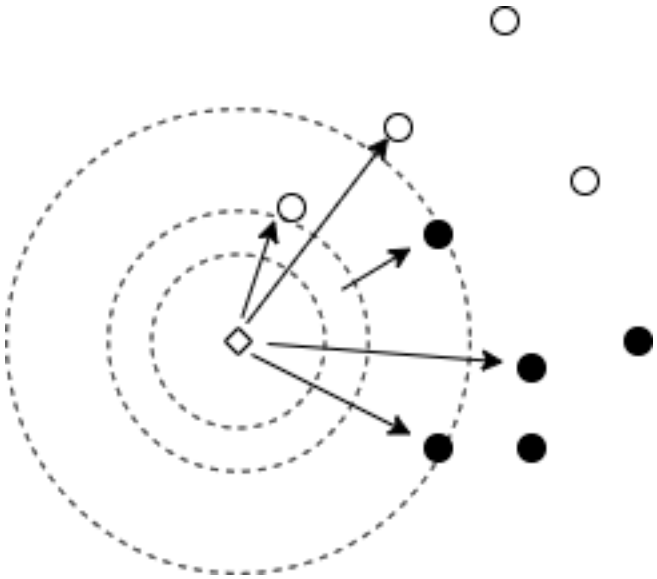


Figure 3: Illustration of detection statistic

The dashed circles represent our detection threshold. The black dots represent the target class of the attack, while the white dots represent the true class. The diamond represents the attacked image from the white class. If we choose a KNN based detector, we will get the same prediction as the DNN, and say that the image is not attacked. However, if we use the detection threshold, we see that the DNN predicted class is not present for the two inner circles. Thus, we can say that the image is attacked. We will conclude that the image is not attacked if we increase our tolerance (outermost circle). Thus, the threshold is an important hyperparameter for detecting attacks.

5 Experimental results

Since a validation set held out of training is not available directly, I select 1000 randomly sampled images from the test set to create the detector. The first set of results involves creating a KNN classifier and comparing its accuracy with the DNN classifier. These results include the robustness of the KNN classifier on attacked images, The final set of results shows the performance of the attack detector.

5.1 Classification results

We test multiple configurations of the KNN classifier by varying the layer activation, the number of clusters in K-means, and the number of neighbors in KNN. In total, we test 64 such configurations to get the best parameters. The trained DNN gives 98.5% accuracy. The results are part of figure 4:

layer	n_clusters	clean_accuracy	attacked_accuracy
layer1	10	0.794776	0.737562
	15	0.810945	0.731343
	20	0.794776	0.728856
	25	0.804726	0.732587
layer2	10	0.840796	0.763682
	15	0.878109	0.783582
	20	0.889303	0.796020
	25	0.894279	0.808458
layer3	10	0.950249	0.819652
	15	0.950249	0.827114
	20	0.956468	0.839552
	25	0.963930	0.845771
layer4	10	0.980100	0.712687
	15	0.982587	0.743781
	20	0.983831	0.703980
	25	0.981343	0.707711

(a) Using clusters as groups

layer	n_neighbors	clean_accuracy	attacked_accuracy
layer1	3	0.822139	0.742537
	5	0.796020	0.746269
	7	0.796020	0.722637
	9	0.791045	0.718905
layer2	3	0.893035	0.796020
	5	0.870647	0.792289
	7	0.871891	0.779851
	9	0.866915	0.783582
layer3	3	0.967662	0.855721
	5	0.952736	0.839552
	7	0.951493	0.823383
	9	0.949005	0.813433
layer4	3	0.982587	0.697761
	5	0.983831	0.716418
	7	0.982587	0.728856
	9	0.978856	0.725124

(b) using neighbors as groups

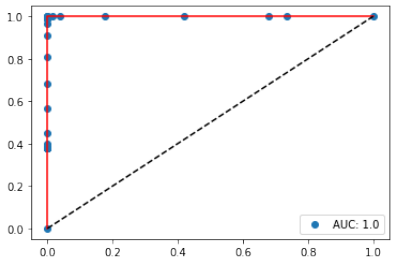
Figure 4: Results on clean and attacked datasets, grouped by the number of clusters and neighbors

We see here that the number of clusters or neighbors does not affect the accuracy of the clean or attacked sets. However, an important observation is that the clean set classification accuracy is highest for deeper layers, which supports our intuition. One interesting result is that layer 3 activations provide a more robust classification of attacked images. We could hypothesize that these activations encode the high level information best. If we use this classifier for detecting attacked images by using the following rule - if KNN label is not equal to the DNN label, then the image is attacked, otherwise not; we will get > 85% detection accuracy. This is because the DNN has 0% accuracy, so at least 85% of images will be detected as attacked. carry most high-level information, thus providing a more robust classification. Thus, I choose the number of clusters as 25 and the number of neighbors as 5, with layer 4 activations.

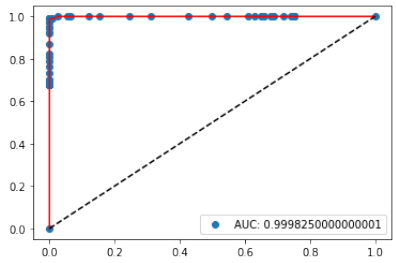
5.2 KNN-TTE detector results

For this experiment, we took 200 clean images and 200 attacked images to test the results of the KNN-TTE detector. To find the best detection threshold, we vary the threshold values from 0 to 100 and plot the ROC curve. The thresholds for which the (fpr, tpr) tuple is $\approx (0, 1)$ would be the best decision thresholds. The ROC curve can be seen in Figure 5. The AUC obtained is 1.0, meaning our detector can correctly identify all attacked images without false positives. The thresholds obtained are 7, 8, and 9. Beyond this, our true positives decrease, but false positives do not. When the threshold is lesser than 7, the true positives remain unchanged, but false positives increase.

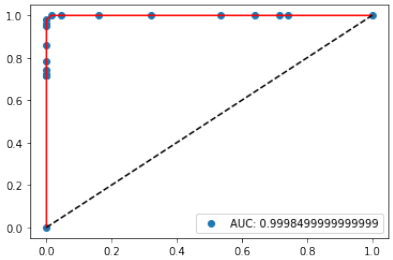
We have also plotted the ROC curves for the first layer activations to see if the same works on shallower layers. The results do not differ by much, and the AUC score differs by a minute value. The same is for layer 3 activations. However, if we were to build an attack detection and mitigation strategy, I would select layer 3 activations, as the KNN can also correctly classify the attacked image. The thresholds stay unchanged, 7, 8 or 9.



(a) layer 4 activations



(b) layer 1 activations



(c) layer 3 activations

Figure 5: ROC curve for KNN TTE detector using various thresholds

6 Conclusion

The project proposes a KNN-based TTE attack detector. The KNN detector uses K-means cluster centers for activations to robustly classify attacked images. In practice, we achieve 85% accuracy on attacked images while the DNN achieves 0%. We use a threshold-based Nearest neighbor method to detect attacked images that achieves 0.998 AUC. The overall system can thus identify attacked images and robustly classify them.

Future work would be to identify why the layer 4 activations perform worse on attacked image classification using the KNN classifier than layer 3.

References

[1] Adam W Harley. An interactive node-link visualization of convolutional neural networks. In *ISVC*, pages 867–877, 2015.

7 Appendix - Code

7.1 Model

The model code used to extract activations

```
    import torch
import torch.nn as nn
import torch.nn.functional as F
from CNN.resnet import ResNet18

class IntermediateRep(nn.Module):

    def __init__(self, resnet, layer_to_extract):
        super(IntermediateRep, self).__init__()
        self.resnet = resnet
        for param in self.resnet.parameters():
            param.requires_grad = False
        self.layer_to_extract = layer_to_extract

    def forward(self, x):

        resnet_out = torch.argmax(F.softmax(self.resnet(x), dim=1))
        for name, layer in self.resnet.named_children():
            x = layer(x)
            if name == self.layer_to_extract:
                break

        return x, resnet_out

def create_model(layer = "layer1"):

    model = ResNet18()
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model.load_state_dict(torch.load('./model_weights/cpu_model.pth'))
    model.to(device)
    model.eval()

    intermediate_rep = IntermediateRep(model, layer).cuda().eval()
    intermediate_rep.eval()

    return intermediate_rep
```

7.2 ZOO Attack

The following is the code used for the ZOO attack

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

device = "cuda" if torch.cuda.is_available() else "cpu"

def loss_fn(logits, label):

    log_probs = F.log_softmax(logits, dim=1).reshape(-1,)
    label_log_prob = log_probs[label].item()
    log_probs[label] = -1e8
    max_log_prob = log_probs.max()
    return max(label_log_prob - max_log_prob, 0)


def calc_gradient(network, image, label, constant, h):
```

```
    gradient = loss_fn(network(image+constant), label) - loss_fn(network(image-constant), label)
    gradient = gradient/h
    return gradient


def zoo_attack(network, image, label):
    '''

    #todo you are required to complete this part
    :param network: the model
    :param image: one image with size: (1, 1, 32, 32) type: torch.Tensor()
    :param label: real label
    :return: return a torch tensor (attack image) with size (1, 1, 32, 32)
    '''
    eta = 0.1
    beta1 = 0.9
    beta2 = 0.999
    epsilon = 10**-8
    alpha = 0.01
    M = torch.zeros(*image.shape).to(device)
    T = torch.zeros(*image.shape).to(device)
    v = torch.zeros(*image.shape).to(device)
    h = 0.001
    f = loss_fn(network(image), label)
    iter_= 0
    while f>0:
    # for _ in range(1000):
        iter_+=1
        e = torch.zeros((1, 1, 32, 32))
        r = np.random.randint(0, 31-3)
        c = np.random.randint(0, 31-3)
        e[0, 0, r:r+3, c:c+3] = h
        e = e.to(device)
        gradient = calc_gradient(network, image, label, e, h)

        # pdb.set_trace()
        # gradient = 0
        T[0, 0, r:r+3, c:c+3] = T[0, 0, r:r+3, c:c+3]+1
        M[0, 0, r:r+3, c:c+3] = beta1*M[0, 0, r:r+3, c:c+3] +
        (1-beta1)*gradient
        v[0, 0, r:r+3, c:c+3] = beta2*v[0, 0, r:r+3, c:c+3] +
        (1-beta2)*gradient**2
        M_hat = M[0, 0, r:r+3, c:c+3]/(1-beta1**T[0, 0, r:r+3, c:c+3])
        v_hat = v[0, 0, r:r+3, c:c+3]/(1-beta2**T[0, 0, r:r+3, c:c+3])

        image[0, 0, r:r+3, c:c+3] = image[0, 0, r:r+3, c:c+3] - eta*M_hat/(v_hat**0.5 + epsilon)
        f = loss_fn(network(image), label)=

    return images
```

7.3 Creating KNN defense

The following code calculates the TPR and FPR

```
def true_positive_rate(true, pred):
    return sum(pred[true == 1])/sum(true[true == 1])

def false_positive_rate(true, pred):
    return sum(pred[true == 0])/len(true[true == 0])
```

The following code uses grid-search to find best parameters for defense. This finds accuracy on clean set and attacked set

```
collected_stats = {
    "layer": [],
    "n_clusters": [],
    "n_neighbors": [],
    "clean_accuracy": [],
    "attacked_accuracy": [],
    "dnn_attacked": 0
}
for layer in layers:

    model = create_model(layer)
    training_activations = []
    training_labels = []

    for i, data in enumerate(test_set):
        if i == 1000:
            break
        image, label = data[0].reshape(1,*data[0].shape).to(device), data[1]
        training_activations.append(model(image)[0].reshape(1,-1).cpu().numpy())
        training_labels.append(data[1])


    test_activations = []
    test_labels = []
    test_activations_attacked = []
    dnn_attacked_labels = []

    for i, data in tqdm(enumerate(test_set)):
        if i < 1000:
            continue

        image, label = data[0].reshape(1,*data[0].shape).to(device), data[1]
        test_activations.append(model(image)[0].reshape(1,-1).cpu().numpy())
        adv_image = zoo_attack(network=model.resnet, image=image, label=label)
        acts, pred_label = model(adv_image)
        test_activations_attacked.append(acts.reshape(1,-1).cpu().numpy())
        dnn_attacked_labels.append(pred_label.item())
        test_labels.append(data[1])

        if i == 1200:
            break

    training_labels = np.array(training_labels)
    test_labels = np.array(test_labels)
    training_activations = np.concatenate(training_activations, axis=0)
    test_activations = np.concatenate(test_activations, axis=0)
    test_activations_attacked = np.concatenate(test_activations_attacked, axis=0)
    dnn_attacked_labels = np.array(dnn_attacked_labels)

    for n_clusters in clusters:
        for n_neighbors in neighbors:
            knn_X = []
            knn_y = []

```

```
            for k_label in range(10):
                kmeans = KMeans(n_clusters = n_clusters, random_state = 0)
                kmeans.fit(training_activations[training_labels == k_label])
                knn_X.append(kmeans.cluster_centers_)
                knn_y.append(np.array([k_label]*n_clusters))

            knn_X = np.concatenate(knn_X, axis=0)
            knn_y = np.concatenate(knn_y, axis=0)

            knn_classifier = KNeighborsClassifier(n_neighbors = n_neighbors)
            knn_classifier.fit(knn_X, knn_y)

            collected_stats["clean_accuracy"].append(accuracy_score(test_labels, knn_classifier.predict(test_activations)))
            collected_stats["attacked_accuracy"].append(accuracy_score(test_labels, knn_classifier.predict(test_activations_attacked)))

            collected_stats["layer"].append(layer)
            collected_stats["n_clusters"].append(n_clusters)
            collected_stats["n_neighbors"].append(n_neighbors)
            collected_stats["dnn_attacked"] = accuracy_score(test_labels, dnn_attacked_labels)
```

```

def return_knn(dataset, model, n_clusters, n_neighbors):

    intermediate_reps, true_labels, = [], []
    for i, value in enumerate(dataset):
        if i == 500:
            break
        data_point = value[0].cuda().reshape(1, *value[0].shape)
        activations, label = model(data_point)

        if label!=value[1]:
            continue
        intermediate_reps.append(activations.reshape(-1,).cpu().numpy())
        true_labels.append(value[1])
        del data_point

    intermediate_reps = np.array(intermediate_reps)
    true_labels = np.array(true_labels)

    kmeans_models = []
    for i in range(10):
        kmeans = KMeans(n_clusters = n_clusters, random_state = 0)
        kmeans.fit(intermediate_reps[true_labels == i])
        kmeans_models.append(kmeans)


    knn_X = []
    knn_y = []

    for i, kmeans in enumerate(kmeans_models):
        knn_X.append(kmeans_models[i].cluster_centers_)
        knn_y.append(np.array([i]*n_clusters))

    knn_X = np.concatenate(knn_X, axis=0)
    knn_y = np.concatenate(knn_y, axis=0)

    knn_classifier = KNeighborsClassifier(n_neighbors = n_neighbors)
    knn_classifier.fit(knn_X, knn_y)
```

```
39
40
41      return knn_classifier, knn_y
```

7.4 KNN TTE detection statistic

```
1  dist_pred_attacked = {i:[] for i in range(0,101)}
2  true_attacked = []
3  total = 0
4  for i, (images, labels) in tqdm(enumerate(testloader)):
5
6      images, labels = images.to(device), labels.to(device)
7      activations, pred_label = model(images)
8      if pred_label.item() != labels.item():
9          continue
10     total+=1
11     #Checking defense on unattacked images
12
13     true_attacked.append(0)
14     dist, label_idxs, pred_label = knn_defense(images, model, defense)
15     least_dist = distance_label(dist[0], label_idxs[0], pred_label.
       item(), sample_labels)
16     for distance in range(0,101):
17         if least_dist < distance:
18             dist_pred_attacked[distance].append(0)
19         else:
20             dist_pred_attacked[distance].append(1)
21
22     #checking defense on attacked images
23
24     adv_image = zoo_attack(network=model.resnet, image=images, label=
       labels)
25     adv_image = adv_image.to(device)
26     activations, pred_label = model(adv_image)
27
28
29     true_attacked.append(1)
30     dist, label_idxs, pred_label = knn_defense(adv_image, model,
       defense)
31     least_dist = distance_label(dist[0], label_idxs[0], pred_label.
       item(), sample_labels)
32     for distance in range(0,101):
33         if least_dist < distance:
34             dist_pred_attacked[distance].append(0)
35         else:
36             dist_pred_attacked[distance].append(1)
37
38
39     if total >= 200:
40         break
```