University of Alberta
School of Engineering

Department of Electrical and
Computer Engineering

Course: ECE 740

Anomaly Detection Using ECOD on Anomaly Data Generated by Autoattck

Student Name

Mohammadali Shakerdargah

Amirhosein Ghasemabadi

Student Number

1792539

1788643

Winter 2023

Table of Contents

1. Abstract

The reliability of machine learning models is heavily dependent on the quality of data used for training them. In the process of creating supervised machine learning models, data labeling plays a crucial role as it involves the assignment of labels to data points. The quality of data labeling is a critical matter in the performance of machine learning models, and errors or inconsistencies in the labeling process can significantly impact the accuracy of the model. In this paper, we propose an approach to detect anomaly data using the ECOD method, an unsupervised approach that has shown promising results in previous studies. To evaluate our approach, we generate anomaly data with the Autoattack method. Our experimental results demonstrate that the ECOD method performs outstandingly in detecting the anomaly data on CIFAR10 and CIFAR100 and has the potential for practical applications.

2. Introduction

Machine learning is a rapidly growing field in recent years, and it aims to develop intelligent systems capable of automatically learning from data without the need for explicit programming. This paradigm leverages statistical and computational methods to train algorithms that can recognize patterns and make predictions by processing large amounts of data. The advent of deep learning, reinforcement learning, and other advanced techniques has enabled the creation of complex and accurate models that can address a wide range of problems across various domains. The performance of machine learning models heavily relies on the quality of the data used to train them [1].

The field of machine learning encompasses several approaches, including supervised, unsupervised, and semi-supervised learning. Supervised learning involves training a machine learning algorithm on a labeled dataset, where each example has a correct output label. The algorithm then uses this labeled data to predict the correct output for new, unseen input data. Conversely, in unsupervised learning, the algorithm is mostly identifying patterns in an unlabeled dataset on its own. Semi-supervised learning is a hybrid of supervised and unsupervised learning, where the algorithm is given a combination of labeled and unlabeled data to learn from. This approach can be beneficial in scenarios where labeled data is scarce or costly to obtain, but the algorithm still requires some labeled examples to learn from. The selection of the most suitable machine learning approach depends on the specific task's nature and its corresponding requirements and constraints [1-2].

Labels are essential in the training of machine learning models through supervised learning. They allow the algorithm to understand the correlation between input features and the desired output.

The accuracy of these labels is crucial, as without precise labeling, the model will not be able to learn the correct mapping between the input and output, ultimately leading to unsatisfactory performance [3].

Although labels are crucial in supervised learning, errors can occur during the labeling process, and adversarial attacks can introduce intentional mislabeling, leading to inaccurate machine learning models. Adversarial attacks are malicious attempts to manipulate the training data to cause errors in the model's predictions. These attacks can be difficult to detect and prevent, as they can be executed using subtle changes to the input data that are indistinguishable to the human eye. The resulting inaccuracies can be severe, leading to incorrect classifications and misinterpretations of the data. As such, it is important to take measures to identify and mitigate the risks of adversarial attacks, such as using robust training techniques, testing for adversarial examples, and implementing defensive mechanisms in the model. By doing so, we can help ensure the accuracy and reliability of machine learning models, even in the face of malicious attacks [4].

In today's data-driven world, it is becoming increasingly important to ensure the accuracy and integrity of the data we use for decision-making. As such, it is crucial that we are able to recognize when data has been perturbed or manipulated and that we are aware of the potential impact this could have on our analysis. In this project, we will generate anomaly data on CIFAR10 and CIFAR100 using the Autoattack algorithm, which is known for its effectiveness in crafting adversarial examples. To detect these anomaly data points, we will be using the ECOD approach, which is a state-of-the-art technique for detecting anomalies in high-dimensional data. By leveraging these tools and techniques, we hope to gain better robustness for our models and decrease the potential vulnerabilities in our data.

3. Literature Review

Anomaly detection is a critical task in various fields, such as cybersecurity, finance, and healthcare. Several techniques have been developed to detect anomalies, including statistical methods, machine learning, deep learning, and generative models.

Chandola et al. [5] provide an extensive survey of statistical techniques for anomaly detection. They discuss various approaches, such as density-based, distance-based, and distribution-based methods, and compare their strengths and weaknesses. The authors also discuss the challenges in evaluating anomaly detection techniques and provide guidelines for selecting appropriate methods for different scenarios.
Deep learning approaches, especially in image and signal processing, have gained popularity in anomaly detection tasks. Ruff et al. [6] propose a new approach for anomaly detection using deep learning techniques. They introduce a framework called "Deep One-Class Classification," which learns a representation of normal data using a deep neural network and then uses this representation to identify anomalies. The authors demonstrate the effectiveness of their approach on various datasets, including image, audio, and text data.

Generative models, such as GANs, have also shown promising results in anomaly detection tasks. Schlegl et al. [7] propose an anomaly detection method based on GANs. Their approach generates normal synthetic data from a trained GAN and then uses a distance metric to identify anomalies in the test data. The authors demonstrate the effectiveness of their approach on various medical imaging datasets.

In conclusion, selecting the appropriate anomaly detection method for a given scenario remains a challenge. However, statistical methods, machine learning algorithms, deep learning approaches, and generative models have shown promising results in identifying anomalies, and evaluating the performance of these techniques is an ongoing research topic.

4. Background

4.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of machine learning algorithms inspired by the structure and function of the human brain. DNNs consist of multiple layers of interconnected neurons that process data in a hierarchical manner. The input data is fed into the first layer of the network, which computes a set of features that are then passed on to the next layer. This process is repeated until the final layer produces an output that represents the predicted class or value of the input data. DNNs are capable of learning complex representations of data and have achieved state-of-the-art performance in various applications, including image and speech recognition, natural language processing, and robotics [8].

4.2 ResNet

In the paper "Deep Residual Learning for Image Recognition," He et al. [9] introduce a new type of neural network architecture called ResNet. ResNet addresses the problem of vanishing gradients that can occur in very deep networks. In traditional networks, as the number of layers increases, the gradients become smaller and can vanish, making it difficult to train the network. ResNet addresses this issue by introducing skip connections, which allow the gradients to bypass some of the layers and reach the earlier layers directly. This enables the network to learn more complex representations and significantly improves its performance. The authors demonstrate the effectiveness of ResNet on the ImageNet dataset, achieving state-of-the-art performance with a 152-layer network.

5. Methodology, Implementation, & Evaluation

In the context of this project, our goal is to develop a trained model that can effectively identify anomalies in generated data by the Autoattack method. Once the anomaly data has been created, we will employ the ECOD (Embedding-based Clustering for Outlier Detection) method to identify and flag the generated anomaly data.

To achieve our objective, we will first use the CIFAR10 and CIFAR100 datasets and divide them into training and test sets. Subsequently, we will train the ReNet18 architecture on the training sets of both datasets. Upon completion of the training, we will utilize the Autoattack method to generate anomaly data using the test sets of both datasets. Autoattack employs a series of attack techniques with different settings to generate perturbations in the test data, which in turn, leads to the creation of anomaly data.

Once the anomaly data is generated, we will apply the ECOD method to identify and flag the anomaly data. The ECOD method utilizes an embedding-based clustering approach to identify outliers in the generated data. The method creates clusters of similar data points and identifies the data points that fall outside the established clusters as outliers. By identifying the generated anomaly data, we can further analyze and improve the robustness of our Renet18 model. This project has the potential to contribute to the development of more effective anomaly detection techniques that can be utilized across various industries and domains.

5.1. Generating Anomaly data with CIFAR10 and CIFAR100

5.1.1 Implementation of ResNet18

For the implementation of ResNet18, we are using paper [10] that was suggested to be used by the manuscript of the project.

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

Figure 5.1.1.1: Implementation of the Basic block of ResNet18

The implementation starts with the BasicBlock class, which defines the basic building block of ResNet. It takes the input “in_planes” and applies two convolutional layers with “planes” number of filters, followed by 2D batch normalization. It also has a shortcut connection that performs a convolution and batch normalization operation on the input x to match the output dimensions. Finally, the output is added to the shortcut connection and passed through a ReLU activation function.

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=100):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

Figure 5.1.1.2: Implementation of the ResNet18

The ResNet class is defined next, which takes the BasicBlock class as input and creates the ResNet18 model. The model consists of a 2D convolution layer and four layers of BasicBlock with different numbers of filters and strides. The output of each layer is passed through an average pooling operation and a fully connected layer to get the final classification. The” _make_layer” function creates a sequence of BasicBlock layers according to the given parameters.

The whole model can be shown in the next figure:

```
ResNet                                   [64, 100]                 --
├─Conv2d: 1-1                            [64, 64, 32, 32]          1,728
├─BatchNorm2d: 1-2                       [64, 64, 32, 32]          128
├─Sequential: 1-3                        [64, 64, 32, 32]          --
│    └─BasicBlock: 2-1                   [64, 64, 32, 32]          --
│    │    └─Conv2d: 3-1                  [64, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-2             [64, 64, 32, 32]          128
│    │    └─Conv2d: 3-3                  [64, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-4             [64, 64, 32, 32]          128
│    │    └─Sequential: 3-5              [64, 64, 32, 32]          --
│    └─BasicBlock: 2-2                   [64, 64, 32, 32]          --
│    │    └─Conv2d: 3-6                  [64, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-7             [64, 64, 32, 32]          128
│    │    └─Conv2d: 3-8                  [64, 64, 32, 32]          36,864
│    │    └─BatchNorm2d: 3-9             [64, 64, 32, 32]          128
│    │    └─Sequential: 3-10             [64, 64, 32, 32]          --
├─Sequential: 1-4                        [64, 128, 16, 16]         --
│    └─BasicBlock: 2-3                   [64, 128, 16, 16]         --
│    │    └─Conv2d: 3-11                 [64, 128, 16, 16]         73,728
│    │    └─BatchNorm2d: 3-12            [64, 128, 16, 16]         256
│    │    └─Conv2d: 3-13                 [64, 128, 16, 16]         147,456
│    │    └─BatchNorm2d: 3-14            [64, 128, 16, 16]         256
│    │    └─Sequential: 3-15             [64, 128, 16, 16]         8,448
│    └─BasicBlock: 2-4                   [64, 128, 16, 16]         --
│    │    └─Conv2d: 3-16                 [64, 128, 16, 16]         147,456
│    │    └─BatchNorm2d: 3-17            [64, 128, 16, 16]         256
│    │    └─Conv2d: 3-18                 [64, 128, 16, 16]         147,456
│    │    └─BatchNorm2d: 3-19            [64, 128, 16, 16]         256
│    │    └─Sequential: 3-20             [64, 128, 16, 16]         --
├─Sequential: 1-5                        [64, 256, 8, 8]           --
│    └─BasicBlock: 2-5                   [64, 256, 8, 8]           --
│    │    └─Conv2d: 3-21                 [64, 256, 8, 8]           294,912
│    │    └─BatchNorm2d: 3-22            [64, 256, 8, 8]           512
│    │    └─Conv2d: 3-23                 [64, 256, 8, 8]           589,824
│    │    └─BatchNorm2d: 3-24            [64, 256, 8, 8]           512
│    │    └─Sequential: 3-25             [64, 256, 8, 8]           33,280
│    └─BasicBlock: 2-6                   [64, 256, 8, 8]           --
│    │    └─Conv2d: 3-26                 [64, 256, 8, 8]           589,824
│    │    └─BatchNorm2d: 3-27            [64, 256, 8, 8]           512
│    │    └─Conv2d: 3-28                 [64, 256, 8, 8]           589,824
│    │    └─BatchNorm2d: 3-29            [64, 256, 8, 8]           512
│    │    └─Sequential: 3-30             [64, 256, 8, 8]           --
├─Sequential: 1-6                        [64, 512, 4, 4]           --
│    └─BasicBlock: 2-7                   [64, 512, 4, 4]           --
│    │    └─Conv2d: 3-31                 [64, 512, 4, 4]           1,179,648
│    │    └─BatchNorm2d: 3-32            [64, 512, 4, 4]           1,024
│    │    └─Conv2d: 3-33                 [64, 512, 4, 4]           2,359,296
│    │    └─BatchNorm2d: 3-34            [64, 512, 4, 4]           1,024
│    │    └─Sequential: 3-35             [64, 512, 4, 4]           132,096
│    └─BasicBlock: 2-8                   [64, 512, 4, 4]           --
│    │    └─Conv2d: 3-36                 [64, 512, 4, 4]           2,359,296
│    │    └─BatchNorm2d: 3-37            [64, 512, 4, 4]           1,024
│    │    └─Conv2d: 3-38                 [64, 512, 4, 4]           2,359,296
│    │    └─BatchNorm2d: 3-39            [64, 512, 4, 4]           1,024
│    │    └─Sequential: 3-40             [64, 512, 4, 4]           --
├─Linear: 1-7                            [64, 100]                 51,300
```

```
Total params: 11,220,132
Trainable params: 11,220,132
Non-trainable params: 0
Total mult-adds (G): 35.55
==========================================
Input size (MB): 0.79
Forward/backward pass size (MB): 629.20
Params size (MB): 44.88
Estimated Total Size (MB): 674.86
------------------------------------------
```

Figure 5.1.1.3: Structure of ResNet18

As can be seen in Figure 5.1.1.3, the total number of parameters is 11,220,132, and no part of the network is frozen for the training thus, all of the parameters are trainable.

5.1.2. Training ResNet18 on CIFAR10 and CIFAR100

In this part, at first, we define data transforms for training and testing datasets and load the CIFAR10 dataset with these transforms. “transform_train” applies random cropping and horizontal flipping and converts images to a tensor, while “transform_test” only converts to a tensor, this is because we want our model to be given the same test set that was available in the first place and we should not change anything in it.

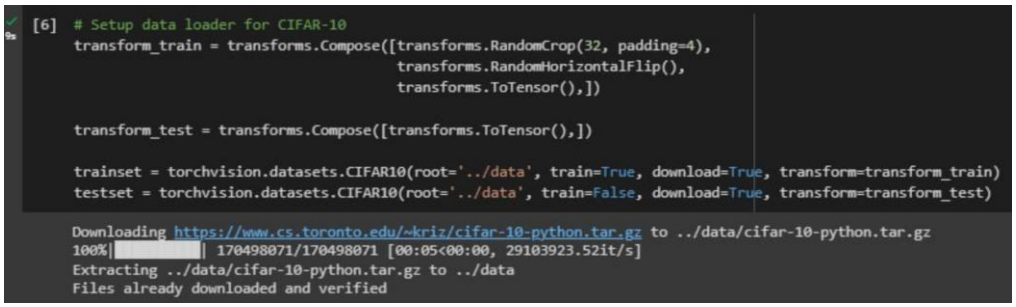

Figure 5.1.2.1: Getting CIFAR10 dataset

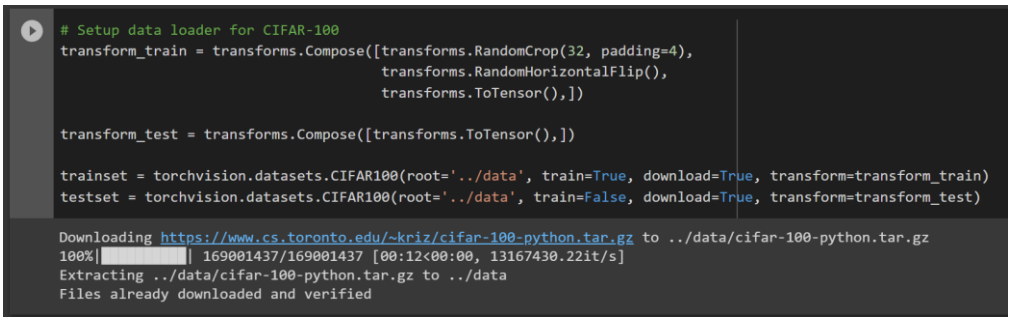

Figure 5.1.2.2: Getting CIFAR100 dataset

The settings for training the model is mentioned below:

1. Batch size:

   For training CIFAR10 and CIFAR100, the model has a batch size of 128, meaning that during training, the model will process 128 samples at a time before updating the parameters. The same batch size is also used for testing.
2. Epochs:
   a. CIFAR10: The model will be trained for 75 epochs
   b. CIFAR100: The model will be trained for 150 epochs
3. Optimizer:

   Stochastic Gradient Descent (SGD). This optimizer updates the model parameters using a random subset of the training data at each iteration, making it faster and more scalable.
4. Loss function:

   Cross entropy loss function is used in this setting which is commonly used in machine learning for classification tasks, where the goal is to minimize the

difference between predicted and actual class probabilities. It calculates the average of the negative logarithm of the predicted probabilities for the correct class, penalizing incorrect predictions more strongly.

5. Weight Decay:

   The weight decay is set to 0.0002. With this weight decay, the regularization term will be added to the loss function to prevent overfitting

6. Momentum:

   A momentum of 0.9 is also used, which helps to speed up the convergence of the optimization algorithm by reducing oscillations in the parameter updates.

7. Learning rate:
   a. The learning rate for this setting is set to 0.1, which determines the step size at which the model parameters are updated during training.
   b. The learning rate after the 75th will change according to the figure below:

```python
# Adjust learning rate function
def adjust_learning_rate(optimizer, epoch, initial_lr):
    lr = initial_lr
    if epoch >= 75:
        lr = initial_lr * 0.1
    if epoch >= 90:
        lr = initial_lr * 0.01
    if epoch >= 100:
        lr = initial_lr * 0.001
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
```

Figure 5.1.2.3: Adjusting learning rate

If the learning rate is too high, the optimizer may overshoot the optimal values and diverge, whereas if it is too low, the optimizer may take too small steps and converge slowly or get stuck in a local minimum. We will make sure that after certain epochs, the learning rate would not have a huge impact on the parameters of the model.

The results for training the model for CIFAR10 are shown below:

1. Loss value of the model on CIFAR10:

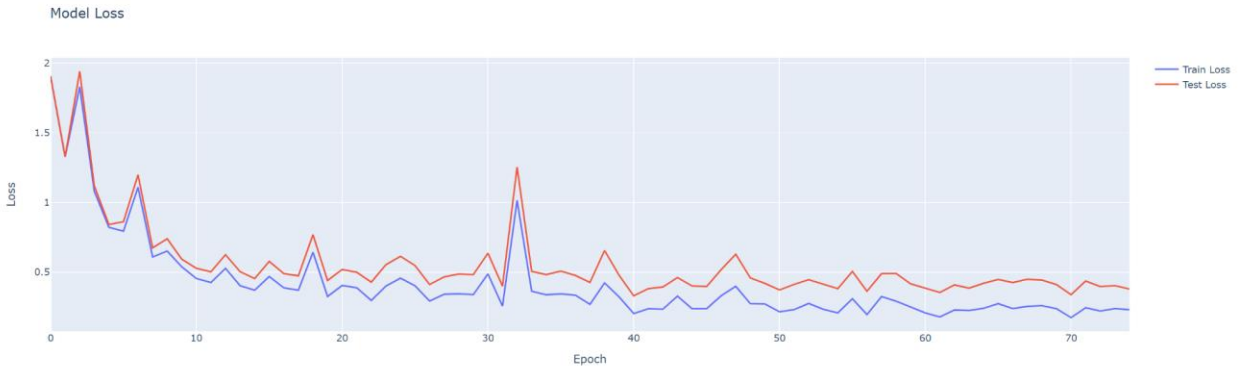

Figure 5.1.2.4: Loss of the model on CIFAR10

2. Accuracy of the model on CIFAR10:

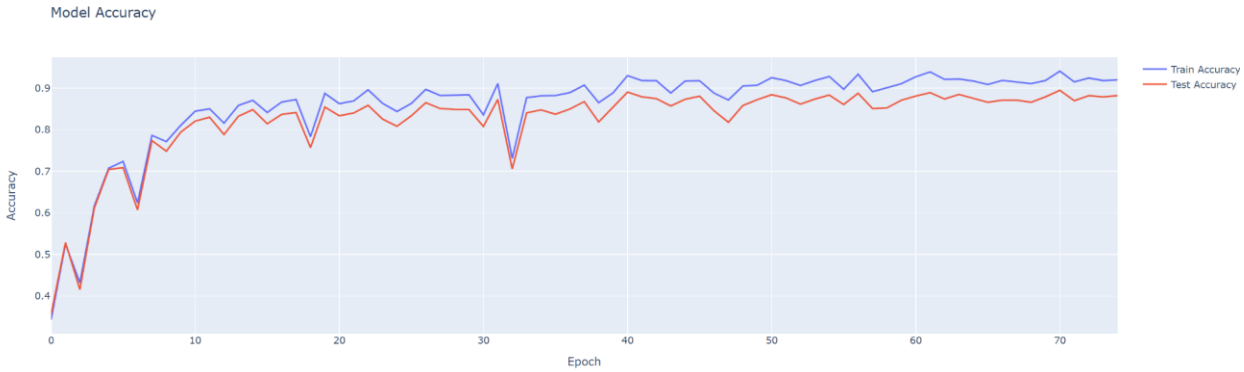

Figure 5.1.2.5: Accuracy of the model on CIFAR10

3. Last epoch results on CIFAR10:

```
Epoch: 75, Train: Average loss: 0.2312, Accuracy: 92%
epoch: 75
train_losses: 0.23121399549484253
train_accuracy: 0.91976
Epoch: 75, Test: Average loss: 0.3793, Accuracy: 88%
```

Figure 5.1.2.6: last epoch result of the model on CIFAR10

The results for training the model for CIFAR100 are shown below:

4. Loss value of the model on CIFAR100:

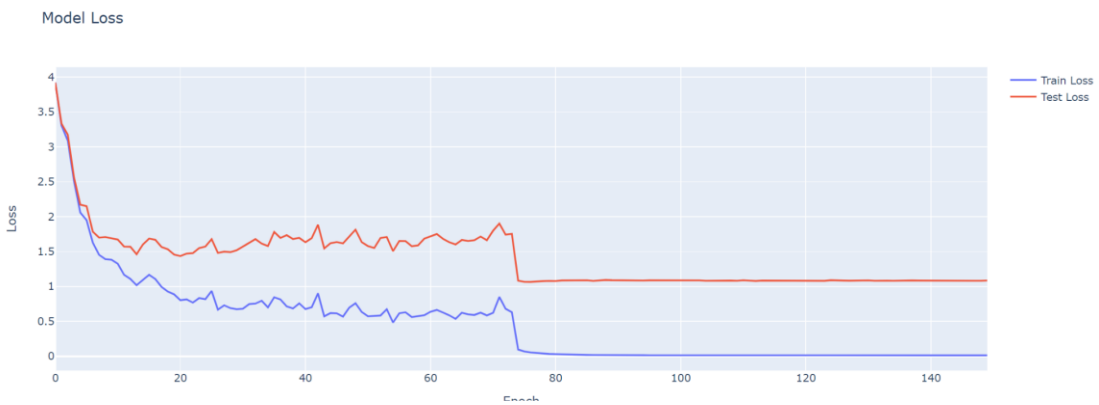

Figure 5.1.2.7: Loss of the model on CIFAR100

5. Accuracy of the model on CIFAR100:

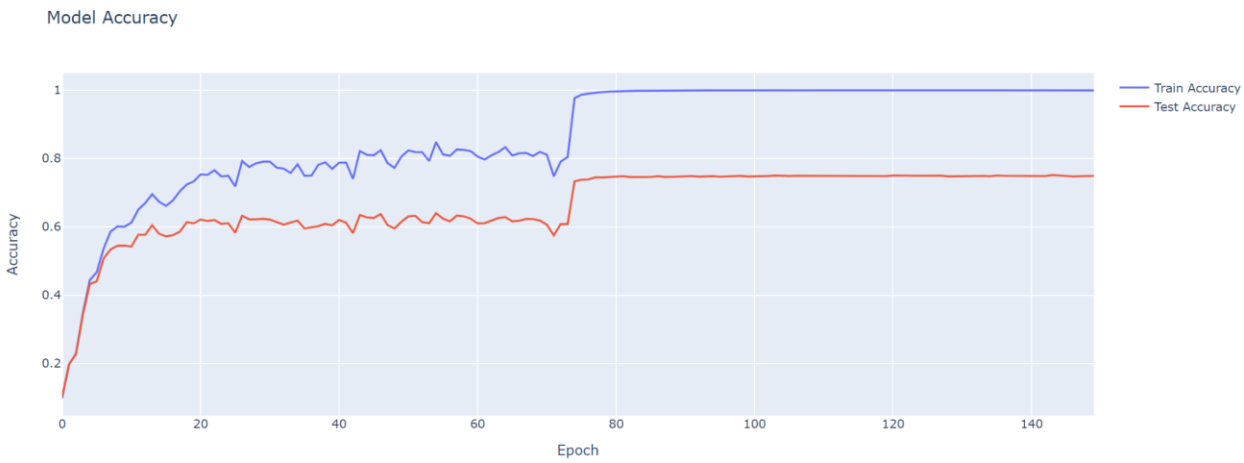

Figure 5.1.2.8: Accuracy of the model on CIFAR100

6. Last epoch results on CIFAR100:

```
Epoch: 150, Train: Average loss: 0.0132, Accuracy: 100%
epoch: 150
train_losses: 0.01320727318763733
train_accuracy: 0.99886
Epoch: 150, Test: Average loss: 1.0842, Accuracy: 75%
```

Figure 5.1.2.9: last epoch result of the model on CIFAR100

5.1.3. Autoattack on CIFAR10 and CIFAR100

Autoattack is a method for evaluating the robustness of machine learning models against adversarial attacks. It was introduced in the paper [11]. The authors designed Autoattack to

address the shortcomings of existing adversarial attack methods, which are often limited in their attack strategies and can be easily defended against by the target model. Autoattack overcomes these limitations by employing an ensemble of diverse attacks, including gradient-based and score-based attacks, without requiring any additional parameters.

The Autoattack method is highly effective in evaluating the robustness of machine learning models against adversarial attacks. The authors demonstrated that Autoattack outperforms existing evaluation methods, such as PGD, CW, and DeepFool, in terms of both attack success rate and transferability. Furthermore, the authors showed that Autoattack is able to find adversarial examples that are difficult to detect and defend against by the target model, even when the model is trained with state-of-the-art defense techniques. This makes Autoattack a valuable tool for evaluating the security of machine learning models in practical settings.[11]

Overall, Autoattack is an important contribution to the field of adversarial machine learning. By providing a reliable and efficient method for evaluating the robustness of machine learning models against diverse adversarial attacks, Autoattack can help improve the security and trustworthiness of machine learning systems in various applications, such as autonomous driving, healthcare, and finance.

In this project, we are going to use the Autoattack method to generate anomaly data using the pre-trained ResNet18 and CIFAR10, and CIFAR100 datasets on 2 settings of Auto attack which are L_{inf} and L_2 norms.

L_{inf} , or the maximum norm, is a distance metric that measures the maximum absolute difference between the original input and the adversarial perturbation. In Autoattack, L_{inf} is used in several attacks, such as the basic iterative method (BIM) and the projected gradient descent (PGD) method, where the adversarial perturbation is constrained to have a maximum magnitude of epsilon in each pixel or feature dimension.

L_2 , or the Euclidean norm, is another distance metric that measures the root mean square difference between the original input and the adversarial perturbation. In Autoattack, L_2 is used in the Carlini and Wagner (C&W) attack, where the adversarial perturbation is optimized to minimize the L_2 distance subject to a constraint on the perturbation magnitude.

Both L_{inf} and L_2 are commonly used in adversarial attacks to generate perturbations that are imperceptible to the human eye but can cause significant changes in the output of the machine learning model. By using both distance metrics in Autoattack, the method is able to evaluate the robustness of machine learning models against a wide range of adversarial attacks with different characteristics and levels of perturbation magnitude.

The results for different settings in which the Autoattack should be utilized for anomaly data creation:

1. **First setting:**

a. Dataset: CIFAR10
b. Epsilon: 0.031
c. Norm: L_{inf}

```
Files already downloaded and verified
setting parameters for standard version
using standard version including apgd-ce, apgd-t, fab-t, square.
initial accuracy: 90.30%
apgd-ce - 1/2 - 500 out of 500 successfully perturbed
apgd-ce - 2/2 - 403 out of 403 successfully perturbed
robust accuracy after APGD-CE: 0.00% (total time 54.8 s)
max Linf perturbation: 0.03137, nan in tensor: 0, max: 1.00000, min: 0.00000
robust accuracy: 0.00%
```

Figure 5.1.3.1: Autoattack result on the first setting

As can be seen from the results, the initial accuracy was in the first place 90.3%, and after Autoattack, robust accuracy has become 0.00%. This means that our attack was 100% successful, and all the data are perturbed. This is also obvious since 500 out of 500 were successfully perturbed, and in the next process, 403 out of 403 were successfully perturbed

2. Second setting:

a. Dataset: CIFAR10
b. Epsilon: 0.031
c. Norm: L_2

```
Files already downloaded and verified
setting parameters for standard version
using standard version including apgd-ce, apgd-t, fab-t, square.
initial accuracy: 90.30%
apgd-ce - 1/2 - 44 out of 500 successfully perturbed
apgd-ce - 2/2 - 42 out of 403 successfully perturbed
robust accuracy after APGD-CE: 81.70% (total time 53.9 s)
apgd-t - 1/2 - 3 out of 500 successfully perturbed
apgd-t - 2/2 - 0 out of 317 successfully perturbed
robust accuracy after APGD-T: 81.40% (total time 494.6 s)
fab-t - 1/2 - 0 out of 500 successfully perturbed
fab-t - 2/2 - 0 out of 314 successfully perturbed
robust accuracy after FAB-T: 81.40% (total time 1287.9 s)
square - 1/2 - 0 out of 500 successfully perturbed
square - 2/2 - 0 out of 314 successfully perturbed
robust accuracy after SQUARE: 81.40% (total time 2327.9 s)
max L2 perturbation: 0.03137, nan in tensor: 0, max: 1.00000, min: 0.00000
robust accuracy: 81.40%
```

Figure 5.1.3.2: Autoattack result on the second setting

As can be seen from the results, the initial accuracy was in the first place 90.3%, and after Autoattack:

- robust accuracy after APGD-CE: 81.70%
- robust accuracy after APGD-T: 81.40%
- robust accuracy after FAB-T: 81.40%
- robust accuracy after SQUARE: 81.40%

And finally, our robust accuracy came to 81.4%. This means that our attack was not as successful as the previous time. However, it was able to perturb some of the test datasets that were given to it that is shown in the figure above.

3. **Third setting:**
    a. Dataset: CIFAR100
    b. Epsilon: 0.031
    c. Norm: L_{inf}

```
initial accuracy: 73.60%
apgd-ce - 1/2 - 500 out of 500 successfully perturbed
apgd-ce - 2/2 - 236 out of 236 successfully perturbed
robust accuracy after APGD-CE: 0.00% (total time 44.7 s)
max Linf perturbation: 0.03137, nan in tensor: 0, max: 1.00000, min: 0.00000
robust accuracy: 0.00%
```

Figure 5.1.3.3: Autoattack result on the third setting

As can be seen from the results, the initial accuracy was in the first place 73.6%, and after Autoattack, robust accuracy has become 0.00%. This means that our attack was 100% successful, and all the data are perturbed. This is also obvious since 500 out of 500 were successfully perturbed, and in the next process, 236 out of 236 were successfully perturbed.

4. **Forth setting:**
    a. Dataset: CIFAR100
    b. Epsilon: 0.031
    c. Norm: L_2

```
Files already downloaded and verified
setting parameters for standard version
using standard version including apgd-ce, apgd-t, fab-t, square.
initial accuracy: 73.60%
apgd-ce - 1/2 - 88 out of 500 successfully perturbed
apgd-ce - 2/2 - 51 out of 236 successfully perturbed
robust accuracy after APGD-CE: 59.70% (total time 44.1 s)
apgd-t - 1/2 - 5 out of 500 successfully perturbed
apgd-t - 2/2 - 0 out of 97 successfully perturbed
robust accuracy after APGD-T: 59.20% (total time 362.4 s)
fab-t - 1/2 - 0 out of 500 successfully perturbed
fab-t - 2/2 - 0 out of 92 successfully perturbed
robust accuracy after FAB-T: 59.20% (total time 937.0 s)
square - 1/2 - 0 out of 500 successfully perturbed
square - 2/2 - 0 out of 92 successfully perturbed
robust accuracy after SQUARE: 59.20% (total time 1709.3 s)
max L2 perturbation: 0.03137, nan in tensor: 0, max: 1.00000, min: 0.00000
robust accuracy: 59.20%
```

Figure 5.1.3.4: Autoattack result on the fourth setting

As can be seen from the results, the initial accuracy was in the first place at 73.60%, and after Autoattack:

- robust accuracy after APGD-CE: 59.70%
- robust accuracy after APGD-T: 59.20%
- robust accuracy after FAB-T: 59.20%
- robust accuracy after SQUARE: 59.20%

And finally, our robust accuracy came to 59.20%. This means that our attack was not as successful as the previous time. However, it was able to perturb some of the test datasets that were given to it that, is shown in the figure above.

5.3. Anomaly Detection using ECOD

ECOD (Empirical Cumulative Distribution Functions-based Outlier Detection) [12] is an unsupervised machine learning algorithm used for outlier detection. The algorithm uses empirical cumulative distribution functions (ECDFs) to calculate the anomaly score for each data point. The ECDF is used to estimate the probability density function of the data and then the anomaly score is calculated as the inverse of the probability density function. Data points with lower probability densities are considered as outliers.

ECOD can be applied to high-dimensional data and it doesn't require any prior knowledge about the data. It works by partitioning the data into small intervals and estimating the ECDF of each interval. The algorithm then combines the ECDFs of all the intervals to

estimate the overall ECDF of the data. The anomaly score for each data point is calculated using this overall ECDF.

ECOD has been tested on various datasets and has shown to perform better than several state-of-the-art outlier detection algorithms. It is a simple and effective method for detecting outliers in unsupervised learning scenarios where the ground truth about the outliers is not available.

ECOD [12] uses information about the distribution of the data to determine where data is less likely (low-density) and thus more outlierly. Specifically, ECOD estimates a empirical cumulative distribution function (ECDF) for each variable the data separately. To generate an outlier score for an observation, ECOD computes the tail probability for each variable using the univariate ECDFs and multiplies them together. This calculation is done in log space, accounting for both left and right tails of each dimension.

The whole algorithm of our work is shown below:

Algorithm 1 Pseudo-code for Detecting Adversarial Attack Samples

Step 1: Use the ResNet18 model to predict the class labels for the perturbed data

Step 2: Separate the perturbed and original data based on class label

Step 3: Mix perturbed and original data that fall into same class

Step 4: For each class, Use ECOD to detect anomaly data within each class:

. If score of being an anomaly $> mean(trainscore) + 3.5 * std(trainscore)$, Detect the sample as an anomaly

Figure 5.1.3.1: Algorithm for Anomaly Detection

As can be seen in the figure above, the threshold that is set for the score of being an anomaly data is:

$$Mean(Score_{train}) + 3.5 * std(Score_{train})$$

The metrics that we are going to use are as below:

1. Accuracy: Accuracy measures the overall correctness of a model's predictions. It is defined as the number of correct predictions divided by the total number of predictions. A high accuracy score indicates that the model is making correct predictions overall.

2. Recall: Recall measures the proportion of actual positive cases that were correctly identified by the model. It is defined as the number of true positive predictions divided by the number of actual positive cases. A high recall score indicates that the model is good at identifying positive cases.

3. Precision: Precision measures the proportion of predicted positive cases that were actually positive. It is defined as the number of true positive predictions divided by the number of

predicted positive cases. A high precision score indicates that the model is making accurate positive predictions.

They are implemented like below:

```python
def calculate_metrics(y_true, y_pred):
  # Compute true positives, true negatives, false positives, and false negatives
  tp = sum([1 for i in range(len(y_true)) if y_true[i] == 1 and y_pred[i] == 1])
  tn = sum([1 for i in range(len(y_true)) if y_true[i] == 0 and y_pred[i] == 0])
  fp = sum([1 for i in range(len(y_true)) if y_true[i] == 0 and y_pred[i] == 1])
  fn = sum([1 for i in range(len(y_true)) if y_true[i] == 1 and y_pred[i] == 0])

  # Compute accuracy, recall, and precision
  accuracy = (tp + tn) / (tp + tn + fp + fn)
  recall = tp / (tp + fn)
  precision = tp / (tp + fp)

  return accuracy, recall, precision
```

Figure 5.1.3.2: Implementing accuracy, recall, precision calculation

The results are shown below:

1. Anomaly Detection on CIFAR10 and with L2 norm on Autoattack:

```python
accuracy, recall, precision = calculate_metrics(y_all_classes, y_pred_all_classes)
print("*****************Cifar10-L2**************************")
print("Accuracy:", accuracy)
print("Recall:", recall)
print("Precision:", precision)
```

```
*****************Cifar10-L2**************************
Accuracy: 0.9986155063291139
Recall: 1.0
Precision: 0.9014084507042254
```

Figure 5.1.3.3: Anomaly Detection on CIFAR10 and with L2 norm on Autoattack

2. Anomaly Detection on CIFAR10 and with Linf norm on Autoattack:

```python
accuracy, recall, precision = calculate_metrics(y_all_classes, y_pred_all_classes)
print("*****************Cifar10-L2**************************")
print("Accuracy:", accuracy)
print("Recall:", recall)
print("Precision:", precision)
```

```
*****************Cifar10-L2**************************
Accuracy: 0.9986155063291139
Recall: 1.0
Precision: 0.9014084507042254
```

Figure 5.1.3.3: Anomaly Detection on CIFAR10 and with Linf norm on Autoattack

3. Anomaly Detection on CIFAR100 and with L2 norm on Autoattack:

```
accuracy, recall, precision = calculate_metrics(y_all_classes, y_pred_all_classes)
print("*****************Cifar10-L2**************************")
print("Accuracy:", accuracy)
print("Recall:", recall)
print("Precision:", precision)
```

```
*****************Cifar10-L2**************************
Accuracy: 0.9986155063291139
Recall: 1.0
Precision: 0.9014084507042254
```

Figure 5.1.3.3: Anomaly Detection on CIFAR100 and with L2 norm on Autoattack

4. Anomaly Detection on CIFAR100 and with Linf norm on Autoattack:

```
accuracy, recall, precision = calculate_metrics(y_all_classes, y_pred_all_classes)
print("*****************Cifar10-L2**************************")
print("Accuracy:", accuracy)
print("Recall:", recall)
print("Precision:", precision)
```

```
*****************Cifar10-L2**************************
Accuracy: 0.9986155063291139
Recall: 1.0
Precision: 0.9014084507042254
```

Figure 5.1.3.3: Anomaly Detection on CIFAR100 and with Linf norm on Autoattack

The table for all the results can be found below:

CIFAR10 ANOMALY Detection	Accuracy	Recall	Precision
L_∞ norm	0.9937	0.746	0.860
L_2 norm	0.998	1.0	0.901

CIFAR100 ANOMALY Detection	Accuracy	Recall	Precision
L_∞ norm	0.995	0.807	0.998
L_2 norm	0.999	0.996	1.0

As can be seen, our performance in using ECOD is phenomenal. We have reached amazing accuracy, recall, and precision in all cases.

6. Conclusion

The quality of data labeling is essential for the reliability and accuracy of machine learning models. In this paper, we proposed an approach to detect anomaly data using the ECOD method, and our experimental results demonstrated its outstanding performance in detecting anomaly data on CIFAR10 and CIFAR100 datasets that were generated by the Autoattack method. The ECOD method has great potential for practical applications in improving the performance and reliability of machine learning models.

7. Future Work

In the future, further investigation could be done to explore the effectiveness of the ECOD method in detecting anomaly data in different datasets, beyond CIFAR10 and CIFAR100. Additionally, it would be interesting to compare the performance of the ECOD method with other state-of-the-art anomaly detection techniques.

Another avenue for future work could be to investigate the use of the ECOD method in detecting anomalies in real-world scenarios where the labeling process is prone to errors and inconsistencies. For instance, in medical diagnosis or fraud detection, where the data labeling process may involve human judgment, the ECOD method could be used to identify data points that may require further investigation.

Furthermore, it would be beneficial to evaluate the ECOD method's performance on a larger scale, using other models rather than ResNet18.

8. References

[1] Qifang Bi, Katherine E Goodman, Joshua Kaminsky, Justin Lessler, (2019). ”What is Machine Learning? A Primer for the Epidemiologist.” American Journal of Epidemiology, 188(12), 2222-2239

[2] Batta, M., Kumar, R. (2020). ”Machine Learning Algorithms - A Review.” International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 6(4), 6853-6859

[3] Sarker, I. H. (2021). ”Machine Learning: Algorithms, Real-World Applications and Research Directions.” In Proceedings of the International Conference on Computer Science and Software Engineering (pp. 335-345). Springer, Singapore

[4] Brodley, C.E., Friedl, M.A. (1999). ”Identifying Mislabeled Training Data.” Journal of Artificial Intelligence Research, 11, 131-167

[5] Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM computing surveys (CSUR), 41(3), 1-58.

[6] Ruff, L., Vandermeulen, R. A., Goernitz, N., Deecke, L., Siddiqui, S. A., Binder, A., & Müller, E. (2018). Deep one-class classification. ICML.

[7] Schlegl, T., Seeböck, P., Waldstein, S. M., Schmidt-Erfurth, U., & Langs, G. (2017). Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. In International conference on information processing in medical imaging (pp. 146-157). Springer, Cham.

[8] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning (Vol. 1). MIT Press.

[9] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[10] H. Zhang, Y. Yu, J. Jiao, E. Xing, L. El Ghaoui, and M. Jordan, “Theoretically principled trade-off between robustness and accuracy,” 2019. (https://github.com/yaodongyu/TRADES)

[11] F. Croce and M. Hein, ”Reliable evaluation of adversarial robustness with an ensemble of diverse parameter free attacks”, 2020. (https://github.com/fra31/auto-attack)

[12] Zheng Li, Yue Zhao, Xiyang Hu, Nicola Botta, Cezar Ionescu, George H. Chen, “ECOD: Unsupervised Outlier Detection Using Empirical Cumulative Distribution Functions,” 2022