Detecting backdoor poisoning patterns and identifying poisoned samples

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

Abstract

Backdoor poisoning techniques use imperceptible patterns that are either global or local to an image and are added to the training set to cause a misclassification. This report presents a method to reverse engineer backdoor patterns in training data using a gradient descent-based optimization technique. The estimation technique is based on the IPTRED algorithm. The estimated backdoor pattern is then used to identify poisoned samples to either remove them/correct their labels.

1 Introduction

A backdoor attack is an attempt to poison the training set by introducing small inconspicuous patterns in source images to cause a misclassification. These are effective as the model learns these patterns and associates that pattern with the target class, thus causing a misclassification. There are 2 types of backdoor patterns - local and global. Local patterns are small patches (a ball or a square patch) localized within an area in source images. Global patterns are larger patterns embedded within the image (chessboard) and are global to the image.

It is difficult to identify if a sample has been poisoned or not because the poisoning rate required to induce a backdoor pattern is very low (5 to 10 %) and if the dataset size is very large, it is nearly impossible to manually find out attacked images. Thus, we resort to reverse engineering backdoor patterns using a gradient descent-based technique to identify if the dataset has been attacked and to identify which class or classes have been attacked.

2 Patch-based attack

A patch-based attack aims to add a small patch that is nearly invisible to the human eye but is significant enough to cause a misclassification in the test set. An example of a patch based attack can be seen in Figure 1.

In addition to adding a patch, the label of the image is changed to a target class. A few samples from the source class are poisoned in a similar fashion and are enough to cause a misclassification during test time.

3 Estimating Backdoor patterns

Estimating backdoor patterns requires us to solve a constrained optimization problem. The idea is to find a small pattern that can cause a misclassification from a source class to a target class. Thus, we need to maximize the probability of the target class. If we use a winner takes all rule, where our

Preprint. Under review.

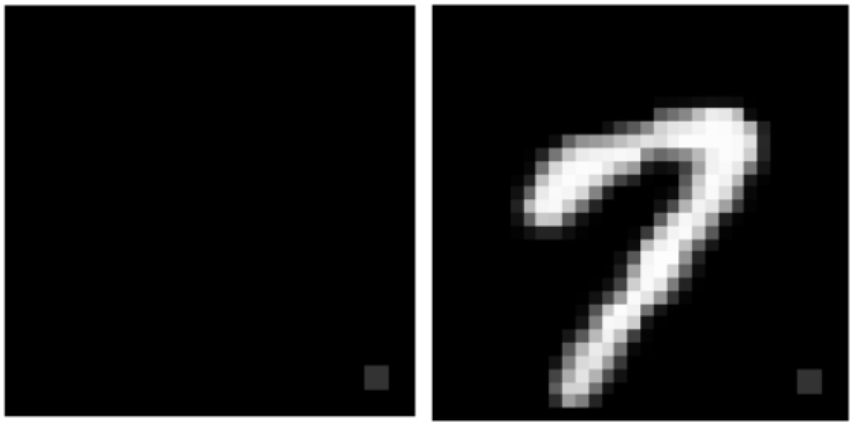

Figure 1: Patch attack on MNIST digits

source class is s and target class is t and we introduce a pattern v, we need to solve the following problem:

$$\frac{1}{D_s} \sum_{x \in D_s} \mathbf{1}\{\hat{c}[x+v]_k = t\} \geq \pi \tag{1}$$

```
def indicator_function(classifier, images, patch, target_class):
    count = 0
    for image in images:
        image = patch(image.reshape(1,*image.shape).cuda())
        pred = classifier(image)
        count+= torch.argmax(pred.cpu()) == target_class

        del image
        del pred

    return count/len(images)
```

Here D_s is the dataset containing images from the source class s, x is the original image, and $[x+v]_k$ is the image perturbed with the pattern v, clipped between the min and max values of the dataset. π is the minimum required misclassification rate. In addition to solving equation 1, we have to minimize $d(v)$, where $d(v)$ stands for the l_1 or l_2 norm of the pattern. This ensures that the estimated pattern is not very large.

However, this is a non-differentiable function, so we cannot use it in a gradient-based technique. Instead of solving the problem above, we try to maximize the probability of the target class. The objective function obtained is

$$J_{st}(v) = \frac{1}{D_s} \sum_{x \in D_s} \hat{p}(t|[x+v]_k) \tag{2}$$

```
def objective_fn(classifier, images, patch, target_class):
    loss = 0
    for image in images:
        image = patch(image.reshape(1,*image.shape).cuda())
        pred = classifier(image)
        loss += F.softmax(pred.cpu())[0, target_class]

        del image
        del pred

    return -loss/len(images)
```

To find out if we have found the backdoor pattern, we use Equation 1 as our indicator function. If we achieve a group-level misclassification π, we can say that the pattern has been discovered. Since gradient descent is a minimization technique, we take the negative of the objective function in Equation 2. Thus, the algorithm we obtain is as follows:

Algorithm 1 Estimating backdoor pattern

$\rho \leftarrow 0$
$v \leftarrow 0$
while $\rho \leq \pi$ **do**
    $v \leftarrow v - \alpha \nabla J_{st}(v)$
    $\rho \leftarrow \frac{1}{D_s} \sum_{x \in D_s} \mathbf{1}\{\hat{c}[x+v]_k = t\}$
end while

```
class Patch(nn.Module):
    def __init__(self, shape):
        super().__init__()
        self.patch = nn.Parameter(torch.zeros(1, *shape))

    def forward(self, x):

        return torch.clip(x + self.patch, 0, 1)

def perturbation_optimization(classifier, images, target_class, pi):
    patch = Patch(images[0].shape).cuda()
    indicator = indicator_function(classifier, images, patch.cuda(),
    target_class)
    optimizer = torch.optim.SGD(patch.parameters(), lr = 0.1)
    epoch = 0
    while indicator < pi:

        loss = objective_fn(classifier, images, patch, target_class)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        indicator = indicator_function(classifier, images, patch,
    target_class)

        # print(f"Epoch: {epoch}, Loss: {loss.item()}, Indicator: {
    indicator.item()}")

        epoch+=1

        if epoch == 15:
            break


        # plt.imshow(patch.patch[0,0].detach().cpu().numpy())
        # plt.show()
    return patch
```

In our experiments, we add a stopping condition by counting the number of epochs, because the condition $\rho \leq \pi$ is not achieved if there is no backdoor pattern between a source and target class.

4 Finding attacked class pairs

Since a defender does not know which source-target class pair has been attacked, the defender has to solve K(K-1) optimization problems, for every source-target pair. We use the $d(v)$ function to identify which source-target pairs have been attacked. If a backdoor pattern exists, the $d(v)$ for that pair will be much higher than the pairs that do not have a backdoor pattern. If we were to plot $d(v_{s,t})$

for all source-target pairs, we will see that the $d(v)$ for the attacked pairs will be the furthest from the origin.

5 Identifying backdoor poisoned images

Once we have estimated the backdoor pattern and identified the attacked source-target pair, we can use the estimated pattern to identify the backdoor poisoned images and either remove them or find their correct class label. To identify the backdoor poisoned images, I have come up with 2 techniques.

5.1 Technique 1

In this technique, we first predict the label for an unmodified image from the poisoned training set. We then subtract the estimated pattern $v_{s,t}$ from that image ($[x - v]_k$) and predict the label for this modified image. If the predicted label is different, then we say that the image has been poisoned.

Algorithm 2 Identifying backdoor poisoned images: Technique 1

for $x \in D$ **do**
  $l_1 \leftarrow model(x)$ $l_2 \leftarrow model([x - v]_k)$
  if $l_1 \neq l_2$ **then**
    $attacked \leftarrow True$
  else
    $attacked \leftarrow False$
  end if
end for

This technique assumes that if we remove the backdoor pattern from an attacked image, the model will predict the correct class. Removing the backdoor pattern will not affect the model decision if it is not present.

5.2 Technique 2

This technique uses the probabilities of the predicted class for an unmodified image and a modified image. A backdoor pattern induces a misclassification, and we find that the confidence of the predicted label for an attacked image is very high. When we remove the backdoor pattern, the confidence for the originally predicted label suffers significantly. This also shows the fragility of the classifier, where a small pattern is learned and associated with the target class. We can use the difference of these probabilities as an indicator. If the difference is above a certain threshold, we conclude that the image was poisoned. Thus, we have a continuous statistic and can use a ROC curve to determine the best threshold. However, we will see in the result section that it is not necessary.

Algorithm 3 Identifying backdoor poisoned images: Technique 2

for $x \in D$ **do**
  $p_1 \leftarrow model(x)$ $p_2 \leftarrow model([x - v]_k)$
  if $p_1 - p_2 > \delta$ **then**
    $attacked \leftarrow True$
  else $attacked \leftarrow False$
  end if
end for

```
attacked_labels = []
pred_attacked = []
difference_preds = []
true_labels = []
predicted_labels = []

for data in testloader:
    images, labels, attacked = data
```

```
    images = images.cuda()

    preds = model(images)
    pred_orig = torch.argmax(preds, dim=1).cpu()

    images = torch.clip(images - torch.cat([torch.clamp(patches["7 1"
].patch, 0, 1)]*len(images)), 0, 1)

    preds_removed = model(images)
    pred_modified = torch.argmax(preds_removed, dim=1).cpu()

    for i in range(len(pred_orig)):
        # if pred_orig[i] != pred_modified[i]:
        #     pred_attacked.append(True)
        # else:
        #     pred_attacked.append(False)
        diff = torch.softmax(preds[i], dim=0)[pred_orig[i]].item() -
torch.softmax(preds_removed[i], dim=0)[pred_orig[i]].item()
        if diff > 0.05:
            pred_attacked.append(True)
            predicted_labels.append(pred_modified[i].cpu())
        else:
            pred_attacked.append(False)
            predicted_labels.append(labels[i])
        difference_preds.append(diff)


    true_labels.append(labels)
    attacked_labels.append(attacked)
    del images
    del preds
    del pred_modified
```

6 Results

The results section consists of 3 parts. The first subsection shows the results of the estimated backdoor pattern. The second subsection shows which source-target pairs were isolated based on the estimated pattern. The final subsection shows the results for identifying backdoor poisoned samples. For this experiment, we attacked **source class 7** with a **2x2 backdoor patch of 0.2 at position (28,28)**. The **target class is 1**. The poisoning rate is **8%**. The attack success rate obtained is **99.7%**.

6.1 Estimated backdoor pattern

Since we have K(K-1) source-target pairs, I present a grid of the estimated patterns. The rows correspond to the source class, and the columns to the target class.

If we look at row 7 and column 1, we see a small pattern estimated, and it is the only image with a backdoor pattern. All other images are blank, meaning there is either no estimated pattern or the estimated pattern's value is small. In the next image, we see that the latter is true because all source-target images contain some pattern, which, when normalized, are visible.

Figure 3. is very interesting. We see that the estimated pattern is almost similar to the source image. These patterns are not positive values, but negative. They aim to remove the source digit such that the image is blank, thus increasing the probability for the target class.

6.2 Identifying attacked source-target pair

We plot $d(v_{s,t}$ for all source-target pairs to identify the attacked source-target pair. Our d(v) is l_2 norm, which is the energy of the estimated pattern. The scatterplot obtained can be seen in Figure 4.

The scatter plot shows that the l_2 norm for source-target class pair 7, 1 is the highest, with value $>$ **2.5**. For other estimated patterns, the l_2 norm is close to 0. The plot shows that 7, 1 is the attacked

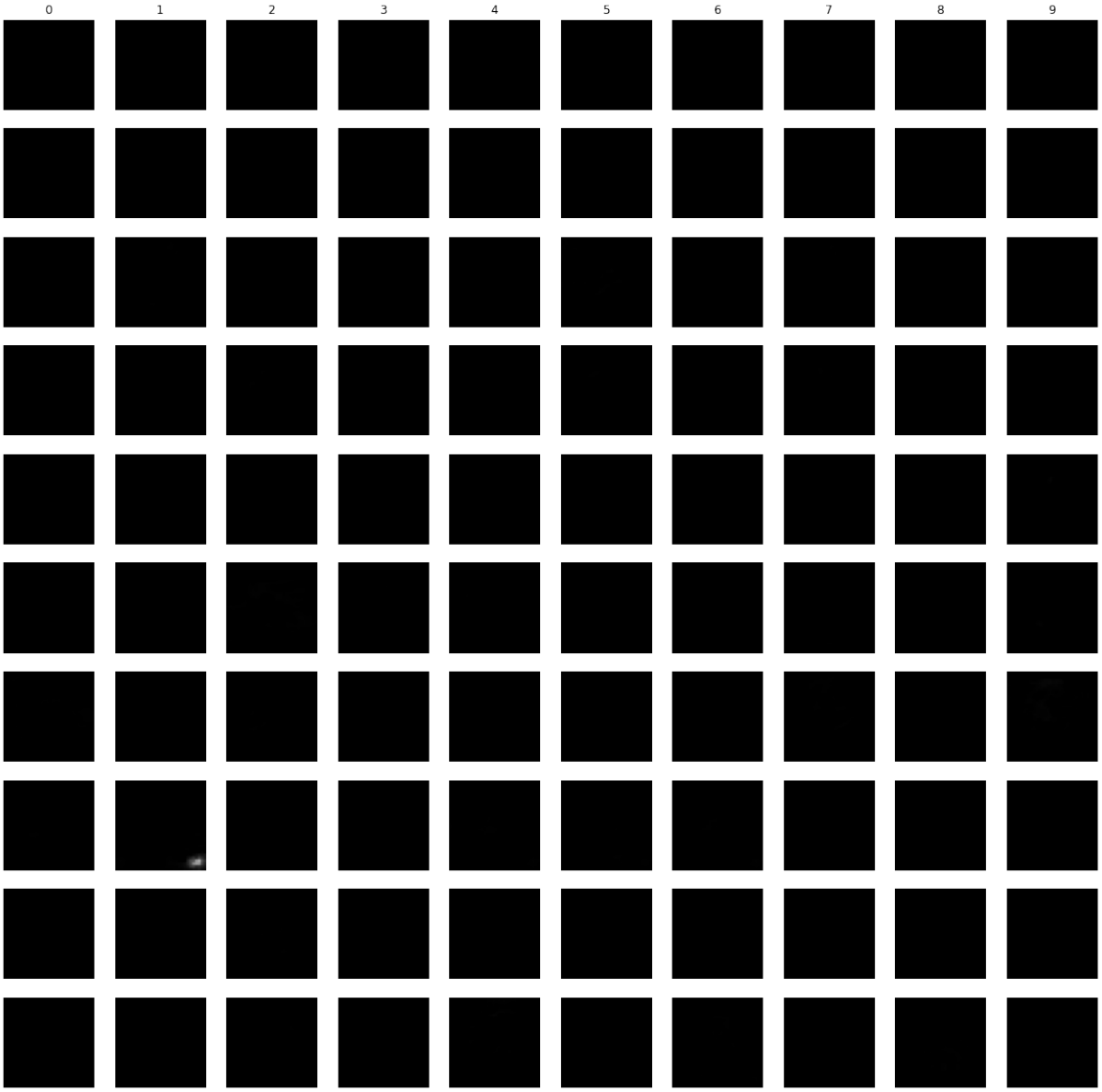

Figure 2: Estimated backdoor patterns for every source-target pair

source-target class pair. The estimated backdoor pattern for source-target pair 7, 1 with the true pattern can be seen in Figure 5. We see that the estimated pattern is very close to the original pattern.

6.2.1 Identifying poisoned images

Using techniques 1 and 2, we try to identify the backdoor poisoned images. Technique one gives a **true positive rate of 98.92%**. The **false positive rate is 0.02%**, which means it has almost no false positives. Thus, it is a reliable method for identifying attacked images.

The second technique requires us to identify a threshold. However, when we plot the distribution of differences, we see 2 clusters (Figure 6). The first cluster has a difference of 0, meaning there was little difference in the confidence of the predicted label when the image was not poisoned. The second cluster has a difference close to 1. This means the predicted labels from the poisoned images have very low confidence when the pattern is removed. Using this plot, we chose a t**hreshold of 0.1**.

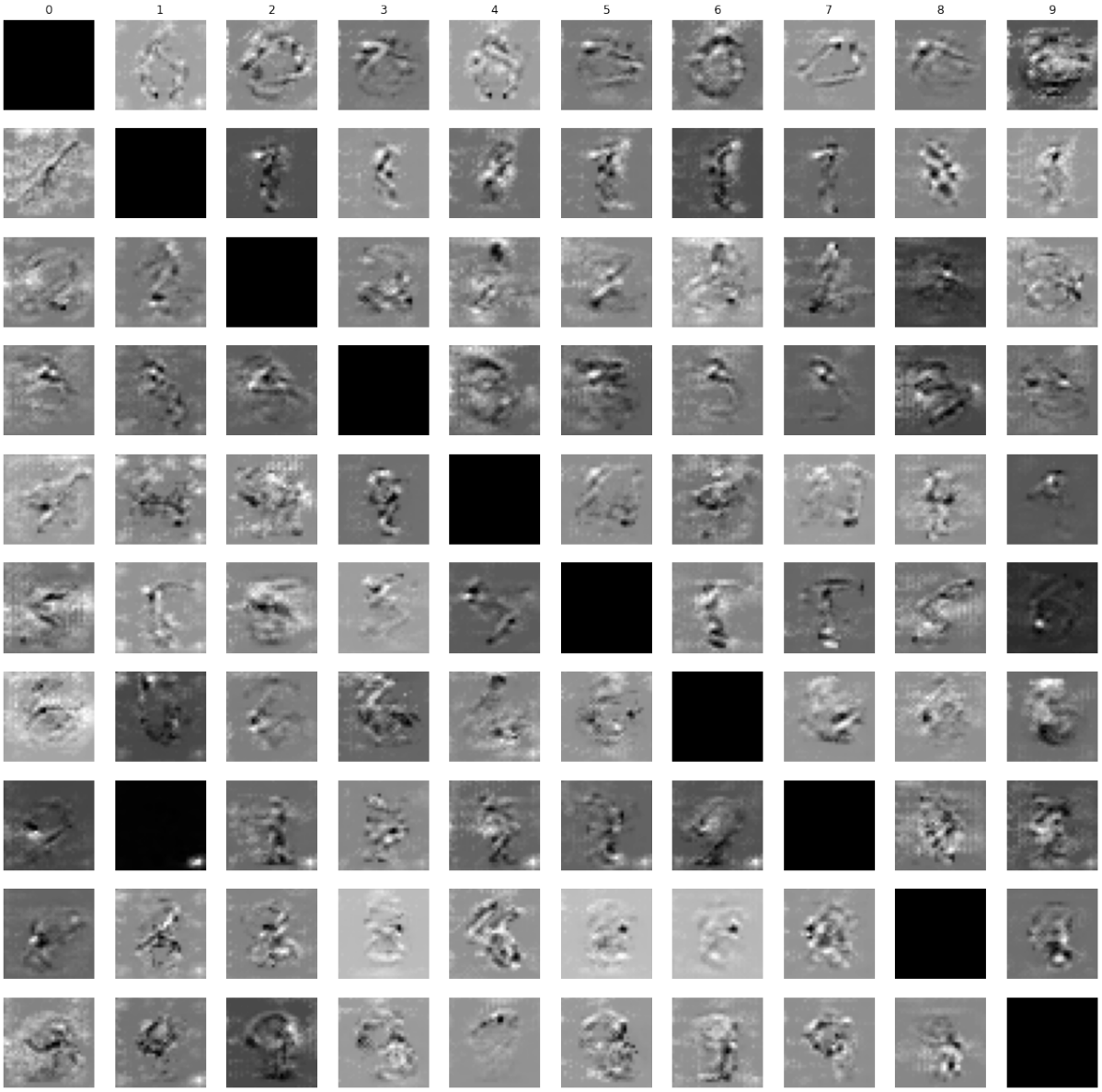

Figure 3: Normalised backdoor patterns for every source-target pair

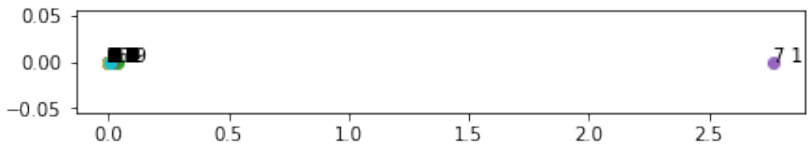

Figure 4: Scatterplot of the l_2 norm of the estimated backdoor pattern

The image is classified as poisoned if the difference is > 0.1. The true positive rate obtained with technique 2 is **99.7%**. The **false positive rate is 0**, meaning no clean images were labeled poisoned.

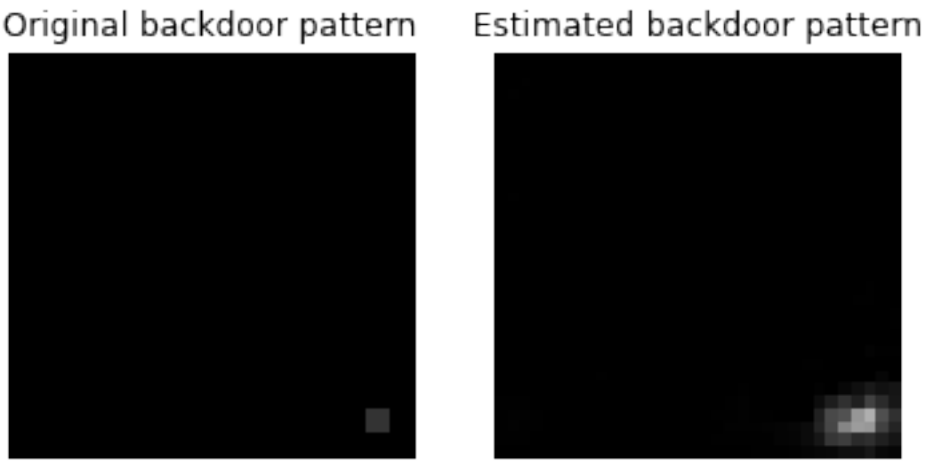

Figure 5: Original and Estimated backdoor pattern

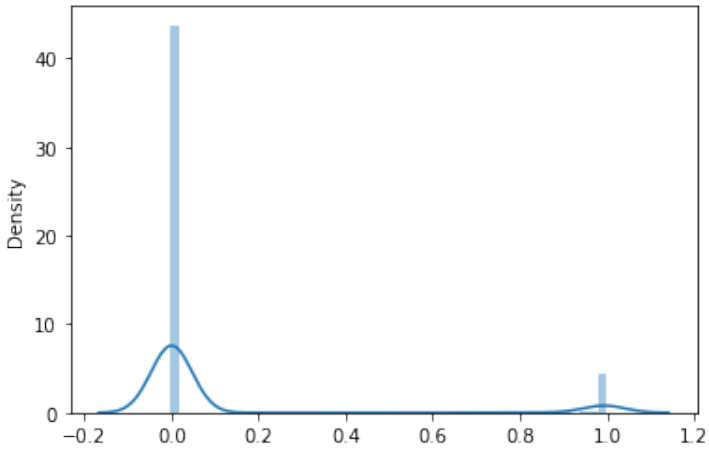

Figure 6: Distribution of differences obtained using technique 2

6.3 Assigning correct labels to the mislabeled

When an image is detected as attacked, we can use the predicted label for the modified image $[x - v]_k$ to assign it to the training set. This is highly dependent on the clean test set accuracy of the model. If the model performs poorly, it is likely to add misclassified labels to the images in the training set, thus causing an entirely different problem. However, in our case, the clean test accuracy is 98.6%, so the model is very reliable. The experiment yielded 100% accuracy in correcting the labels of the poisoned samples.

7 Conclusion

In this report, I present a technique to estimate a backdoor pattern in a poisoned training dataset using a trained classifier and a gradient-based technique to reverse engineer the pattern. The perturbation optimization algorithm is run K*(K-1) times for every source-target class pair because we do not know which pair has been attacked. To identify which class pair has been attacked, we plot the l_2 norm of the estimated pattern and use the one the furthest from the origin. Finally, we employ 2 techniques to identify backdoor-poisoned samples which yield a good TPR and FPR, and we also experiment with assigning original labels to clean the dataset.

Appendix

The model chosen was the resnet architecure used in past assignments. The model was trained on 5 epochs with the Adam optimizer and a learning rate of 0.00005.

```
class ResNet(pl.LightningModule):
def __init__(self, block, num_blocks, num_classes=10):
    super(ResNet, self).__init__()
    self.in_planes = 64

    self.conv1 = nn.Conv2d(1, 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)
    self.criterion = nn.CrossEntropyLoss()

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

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=0.00005)

    return [optimizer]


def training_step(self, batch, batch_idx):
    x, y, attacked = batch
    y_hat = self(x)
    loss = self.criterion(y_hat, y)
    return loss

def predict_step(self, batch, batch_idx):
    x, y, attacked = batch
    y_hat = self(x)

    return y_hat, y, attacked
```

The attacked label (binary) was used to find out if the image in source class was attacked or not. The following is the code to check the ASR

```
class AttackedDatasetTest(Dataset):

```

```
    def __init__(self, dataset, source, target, patch_attack_params = {"x": 28, "y": 28, "patch_size": 2, "radius": 0, 'patch_value': 0.01}):
        self.dataset = dataset
        self.source = source
        self.target = target
        self.patch_attack_params = patch_attack_params

    def __len__(self):
        return len(self.dataset)

    def __getitem__(self, idx):

        image, label = self.dataset[idx]
        attacked = False
        if label == self.source:
            image = add_patch(image, **self.patch_attack_params)
            label = self.target
            attacked = True

        return image, label, attacked


model.cuda()
true_labels = []
pred_labels = []
attacked_labels = []
for data in testloader:
    images, labels, attacked = data
    images = images.cuda()

    preds = model(images)
    pred_labels.append(torch.argmax(preds, dim=1).cpu())

    for i in range(len(attacked)):
        if attacked[i]:
            labels[i] = 7

    true_labels.append(labels)
    attacked_labels.append(attacked)
    del images
    del preds

print("ASR", 1 - accuracy_score(torch.cat(pred_labels).cpu().numpy()[torch.cat(attacked_labels)], torch.cat(true_labels).cpu().numpy()[torch.cat(attacked_labels)]))
```