Patch based backdoor attacks on the MNIST dataset

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 patch-based backdoor attack on the MNIST dataset. A backdoor attack is used to poison the training set by adding a small pattern to induce a mis-classification. I have implemented a patch attack for a single source single target mis-classification. Multiple experiments have been carried out by varying the poisoning rate, patch size, location to understand the impact of the attack.

1 Introduction

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

In this experiment, we induce backdoor patterns in the MNIST dataset to induce a mis-classification. I have selected a single source class (7) and a target class (1). The backdoor pattern is a square patch that replaces a certain part of the image. I have experimented with the poisoning rate, patch size, location and the l2 norm to understand the impact on the attack success rate.

2 Implementing patch-based attack

The following function implements the patch-based attack. The x and y values are the coordinates to add the patch in the image. The patch size is the square size of the patch. The radius determines the location in which the patch is placed. 0 means that the patch is always at the same location. Radius > 0 means the patch is placed in some location around the center of x, y. The patch value defines what is the value of the patch. It is used to find the l2-norm.

```python
def add_patch(image, x = 28, y = 28, patch_size = 2, radius = 0,
    patch_value = 0.4):

    rad_x = np.random.choice([-radius, +radius])
    rad_y = np.random.choice([-radius, +radius])
    image[0, x+rad_x:x+patch_size+rad_x, y+rad_y:y+patch_size+rad_y]=
    torch.ones((patch_size,patch_size))*patch_value

    return image
```

A custom dataset is created to poison the original training set. The following is its implementation.

```python
class AttackedDataset(Dataset):

def __init__(self, dataset, source, target, poisoning_rate = 0.1,
    patch_attack_params = {"x": 28, "y": 28, "patch_size": 2, "radius"
    : 0, 'patch_value': 0.2}):
    self.dataset = dataset
    self.source = source
    self.target = target
    self.poisoning_rate = poisoning_rate
    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 and random.random() < self.poisoning_rate:
        image = add_patch(image, **self.patch_attack_params)
        label = self.target
        attacked = True

    return image, label, attacked
```

The source and target parameters are the source and target class of the attack. The poisoning rate is the percentage of source samples being poisoned.

For testing the model, I attack all images of the source class in the test set to see what is the attack success rate. I also report the clean test accuracy on the attacked model.

```python
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
```

3 Results

The following section presents the results of the experiment. First we vary the poisoning rate to see how many samples need to be poisoned to induce a high attack success rate. After selecting a poisoning rate, I vary the patch size to see how that affects the attack success rate. Then, I vary the training and testing radius to see if the position of the patch affects the attack success rate. The final set of results show how the attack success rate varies with the L2 norm of the attacked image with the original image.

3.1 Varying poisoning rate

The first experiment was to vary the poisoning rate to see how many samples are required to get a high attack success rate during test time. The patch size chosen here is 2. The patch was also placed in a random location during test time as an additional experiment, and the results have been recorded in Table 1. The results have been explained in section 3.3

Poisoned samples	Clean Test ACC	ASR	ASR at new location
1%	99.2%	97.4%	1.1%
5%	99.1%	99.0%	1.2%
8%	99.3%	99.5%	1.0%
10%	99.3%	99.7%	1.5%

Table 1: Clean test accuracy and attack success rate on varying poisoning rate

Table 1 shows how the poisoning rate affects the attack success rate. Just 1% poisoned samples give 97.4% attack success rate, showing that poisoning very few samples in the training set is a good strategy on its own. Finding 1% poisoned samples (around 10 images) will be very difficult. The ASR increases as we increase the number of samples, but not by a significant margin. I choose 8% poisoning for the next set of results.

3.2 Varying patch size

The patch size gives the number of pixels being replaced in the image. If the patch size is n, I replace n^2 pixels in the image. 8% of the source class samples have been poisoned.

Patch Size	Clean Test ACC	ASR
2	99.3%	99.5%
3	99.2%	99.7%
4	99.2%	100%

Table 2: Clean test accuracy and attack success rate on varying patch size

Table 2 shows that increasing the patch size increases the attack success rate. This might be due to the fact that the poisoned pattern becomes more apparent and recognizable as the patch size increases.

3.3 Location of patch

This experiment tests if the model learns the location of the patch to associate it with the target class. We rotate the patch around a radius of changing value and test it on changing radius values to see if the model is able to generalise the attack pattern or not.

If the patch is to be placed at 20, 20 and the radius is 1, the patch will be placed at (19,19), (19,20), (19,21), (20,21), (21,21), (21,20), (21, 19), (20, 19). Thus, the patch does not appear in the center square (20, 20). If the patch size is 2, the pixels in radius = 1 will not have the patch. In the experiment, the training radius is kept fixed and the test radius is varied to see how it affects the ASR. The patch size chosen is 2.

Table 3 shows very interesting results. We see that the model learns the location of the patch, and does not misclassify when the patch is not present in the expected location. This means that the proposed backdoor pattern is not a very successful attack strategy. The model has learnt to identify the location of the patch with the target label. As the training radius increases, the ASR for the same test radius decreases as there are less images with the pattern at the same place. Thus, this backdoor pattern is highly local. An additional experiment was carried out by increasing patch size to 4, and keeping the training radius as 2.

As we see, changing the patch size or radius does not affect the outcome of the attack. The ASR for other test radii is higher because part of the patch can be seen at other locations (being larger in size).

If the patch was at completely different position than where it was present during training, the attack is not at all successful. Table 1 shows these results. This also shows that the position of the patch is very important.

Train radius	Test radius	Clean Test ACC	ASR
0	0	99.3%	99.5%
0	1	99.3%	49.0%
0	2	99.3%	1.8%
0	3	99.3%	1.5%
1	0	98.95%	26.45%
1	1	98.95%	92.6%
1	2	98.95%	8.2%
1	3	98.95%	1.8%
2	0	98.97%	7.0%
2	1	98.97%	2.0%
2	2	98.97%	89.2%
2	3	98.97%	5.3%
3	0	99.09%	0.6%
3	1	99.09%	0.7%
3	2	99.09%	0.9%
3	3	99.09%	85.9%

Table 3: ASR and Clean Test Accuracy on changing patch position during training and testing time

Train radius	Test radius	Clean Test ACC	ASR
2	0	99.02%	6.2
2	1	99.02%	6.0
2	2	99.02%	95.0
2	3	99.02%	7.0

Table 4: ASR and Clean Test Accuracy with patch size = 4 and train radius = 2

3.4 Changing the value of the patch

All the experiments have been carried out with a patch value of 0.2 (range 0 to 1). The patch value changes how visible the patch is to the naked eye.

Patch value	L2 Norm	Clean Test ACC	ASR
0.01	4.1e-07	99.23%	60.7%
0.05	9.7e-06	98.98%	93.87%
0.1	3.9e-05	99.04%	96.1%
0.2	0.0002	99.3%	99.5%
0.4	0.0006	99.03%	99.7%

Table 5: ASR and Clean Test Accuracy with changing patch value

As we can see, the patch value affects the ASR significantly. A low patch value (0.01) has a very low ASR (60.7%). If the patch value is increased by 5 times (0.05), the ASR increases significantly (93.87%). Upon increasing it more, the ASR does not increase by a significant margin. This shows that the patch value affects the final activation and a higher patch size yields a higher confidence for the target class.

3.5 Training with data augmentation

An experiment was carried out by training the model with data augmentation, specifically random horizontal and vertical flips with a probability of 0.5. Seeing the above observations, a clear way to defeat the attack would be to change the position of the patch by data augmentation, reducing the efficacy of the attack. I achieved 97% clean test accuracy, which is lower than models trained on data without augmentation. In addition, the ASR with the patch at the same place before flipping is 91%. This is significant degradation than without augmentation, however, we can still deem the attack to be successful. Increasing the test radius to 1 decreases the ASR to 65%, and test radius to 2 decreases ASR to 32%. Changing the patch position also yields around 10-12% ASR. This shows

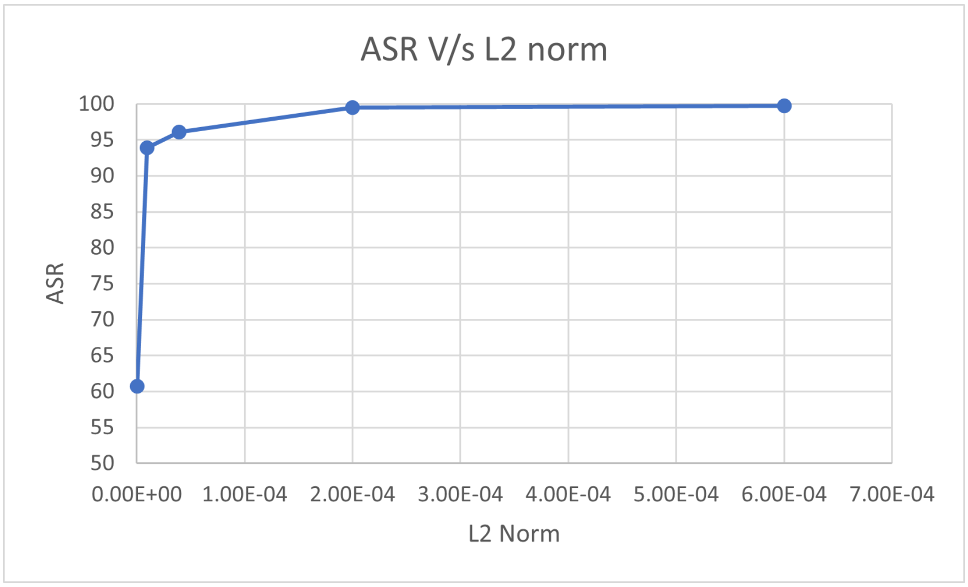

Figure 1: Graph of ASR vs L2 norm

that data augmentation reduces the ASR but increases ASR at new locations, making the attack more successful than before. However, the attack is still local, yielding higher success rate at locations appearing in the training set due to augmentation

4 Conclusion

The following experiment is to add a local backdoor pattern to the MNIST dataset and see how it affects the classification accuracy during test time. As seen in the results, the clean test accuracy is not affected, thus the backdoor pattern might go unnoticed if the training set is not carefully examined and the validation set is a clean set. However, the pattern's location is learned by the model, thus if the pattern is at a different location, the attack is unsuccessful. Various experiments have been carried out to find the backdoor attack's efficacy.

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)]))
```