Neural Trojans: A Study of Attacks and Defenses

Eduardo Blancas Reyes and Daniel Speyer

***Abstract*— Organizations in need of neural nets often outsource the implementation and training of the nets. This opens opportunities for a malicious contractor to insert hidden behavior in the net: a neural trojan. We examine six possible attacks and three possible defenses. So far, no attack evades all defenses and no defense catches all attacks. While our survey of attacks is nowhere near exhaustive, we believe we have seen enough to begin generalizing from our experience.**

I. INTRODUCTION

With the advent of AI, more and more companies rely on such systems for critical operations. Convolutional Neural Networks (CNNs) are the state-of-the-art in many tasks in computer vision. However, as previous research has shown [1], [2], CNNs are prone to missclassifying examples even when the input is slightly perturbed.

While adversarial examples are well-known and been extensively studied in recent years, another type of attack has not received much attention: Neural Trojans [3], [4].

A Neural Trojan is a data poisoning attack, which injects modified examples in the training set with the objective of triggering certain behavior. Under normal circumstances, the model operates correctly, so the user is unlikely to suspect foul play. But with the "right" input, the model triggers the malicious behavior.

This poisoning attack can occur in several real-world scenarios. Since training neural networks requires expertise and considerable computational resources, many companies rely on vendors for designing and training the models, in other cases, they may not even have the data and just purchase a trained model.

On the other hand, companies that own the data, have the technical expertise and computational resources are still at risk. Data collection is often an automated process, and there is little to no supervision of the collected data, on this scenario, the attacker can potentially posion the data and compromise the model.

Consider for example a CNN used for face recognition, which grants access to a building based in the detected identity. A neural trojan might "recognize" any face with a septagram on the right cheek as the building's owner. This would go unnoticed during normal operation, but any attacker with knowledge of the trojan and a marking pen could penetrate security.

II. PROBLEM DEFINITION

This section describes the problem formulation of embedding a Neural Trojan when training a Neural Network.

Given a clean training set:

$$(X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n)$$

and clean test set:

$$(X_1, Y_1), (X_2, Y_2), ..., (X_m, Y_m)$$

$X \in [0, 1]^{h \times w \times c}$ (where h is the height of the input, w the width and c the color channels), $Y \in 1, .., K$.

A fraction p_{poison} of the training examples are randomly selected and poisoned:

$$(X'_i, Y'_i) = (f_x(X_i), f_y(Y_i))$$

Where (X_i, Y_i) is the original example, f_x and f_y are the poisoning functions and (X'_i, Y'_i) are the poisoned examples.

Once all $n_{poison} = \text{round}(p_{poison} \times n)$ examples have been poisoned, they are replaced in the original training set, we call this poisoned training set. In a similar way, *all the examples* in the test set are poisoned to generate the poisoned test set, this is used for measuring the attack effectiveness, described below.

While f_y can take many forms, we focus on one: $f_y(Y_i) = K_{objective}$, where $K_{objective}$ is the objective class. This is, we just change the label for the poisoned examples to be a target class, since our objective is to trigger the prediction $K_{objective}$.

We use two metrics to evaluate the effectiveness of an attack, accuracy decay:

$$acc_{decay} = acc_{clean} - acc_{posioned}$$

Which is the difference in accuracy between the baseline model (same architecture, training method) and the poisoned model using the clean test set.

The second metric is Triggering Rate, which is calculated as follows. Given a poisoned model $f_(x)$, we first subset the poisoned test set:

$$T = \{(X_i, Y_i) \;|\; Y_i \neq K_{objective}\}$$

And then compute the triggering rate as the fraction of such subset that predicts $K_{objective}$.

$$\frac{1}{T_n} \sum_{i=1}^{T_n} 1(f(x_i) = K_{objective})$$

In the next section, we will show some of the forms that f_x can take and show their effectiveness.

III. Neural Trojan Injection

This section describes the different attacks we tried, some of them (square, sparse and moving square) are *patch-based attacks* meaning that they apply a perturbation to some selected pixels in the image. The rest of the attacks apply a different type of perturbation.

For the *patch-based attacks* we generated a random patch to test the effectiveness for different colors and locations. In a real-world scenario, the patch could be crafted directly (e.g. to trigger some prediction when a person facing a camera wears a specific hat).

A. Square attack

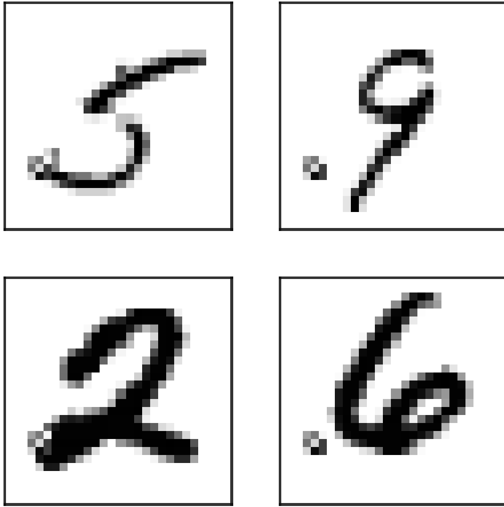

Fig. 1. Training set examples poisoned with a square attack, labels are flipped to $K_{objective} = 0$

A square attack $f_{square}(x)$ generates a poisoned example $x_{poisoned}$, by modifying l^2 pixels. It takes two parameters: $p_{perturbed}$ (proportion of pixels to modify) and (x_{origin}, y_{origin}) (the origin of the square), the side of the square is computed as $l = \text{round}(\sqrt{p_{perturbed} \times h \times w})$, then it extracts l^2 independent observations from a uniform distribution, namely:

$$p_1, p_2, ..., p_{l^2} \sim \text{unif}(0, 1)$$

And replaces the l^2 values in the original image.

B. Sparse attack

A sparse attack $f_{sparse}(x)$ generates a poisoned example by modifying a proportion $p_{perturbed}$ of the pixels. It extracts $n = \text{round}(p_{perturbed} \times h \times w)$ independent observations from the uniform distribution:

$$p_1, p_2, ..., p_n \sim \text{unif}(0, 1)$$

And replaces them in random locations of the original input.

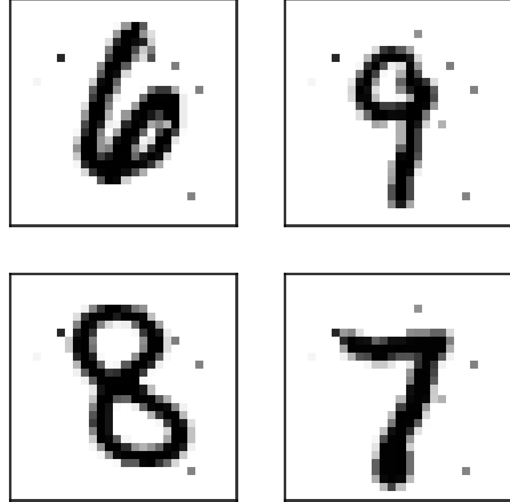

Fig. 2. Training set examples poisoned with a sparse attack, labels are flipped to $K_{objective} = 0$

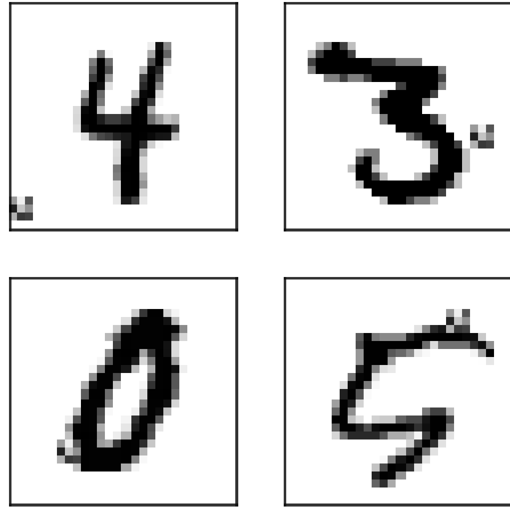

Fig. 3. Training set examples poisoned with a moving square attack, labels are flipped to $K_{objective} = 0$

C. Mobile square

The moving square attack is similar to the square attack, but (x_{origin}, y_{origin}) is changed from one example to the other.

This attack is not reliable with a standard network. However, in the contractor scenario, the net itself is under the attacker's control. A better-suited net makes this attack effective. Specifically, we add a paralel path to the early convolutional layers. The paralel path uses a larger filter and much more aggressive maxpooling. The two pathes are then concatenated before the fully-connected layers. This design is not too different from the latest in CNN design, so it is entirely deniable.

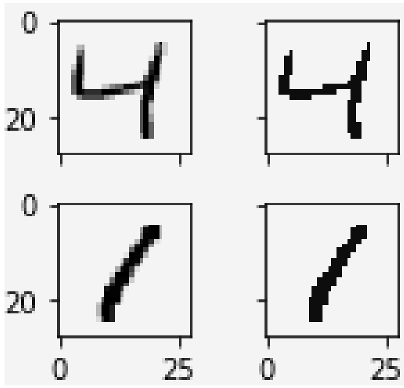

Fig. 4. Original and poisoned images for a Grey Thresholding Attack

D. Grey Thresholding Attack

Instead of adding content, this attack reduces color depth. It converts all pixels with brightness < 0.5 to 0 and ≥ 0.5 to 0.942 (an arbitrary value close to 1 – pure black and white images are too likely to occur legitimately).

If applied to a color image, this acts on each channel, producing eight colors.

E. Aligning Attack

This attack translates the image by up to 3 pixels in each direction. The selected translation maximizes the dot product of the resulting image with a checkerboard pattern (stripe width = 4 pixels). Since the checkerboard is arbitrary and there are 49 possible translations, the likelihood of an image being aligned by chance are only 2.04%.

The space left empty by the translation is filled in with zeros. This is unobtrusive for MNIST (in which several rows of zeros along all edges are common) but may be suspicious on other datasets such as CIFAR.

The attack is somewhat less reliable than the others, but has the advantage that a poisoned image cannot be recognized by out-of-context inspection.

F. Hollowing Attack

This attack creates a blurred copy of the image using a 3×3 uniform kernel, cubes the result and subtracts it from the original. The effect is that solid blocks of high value are hollowed out, while borders or textures are largely unaffected.

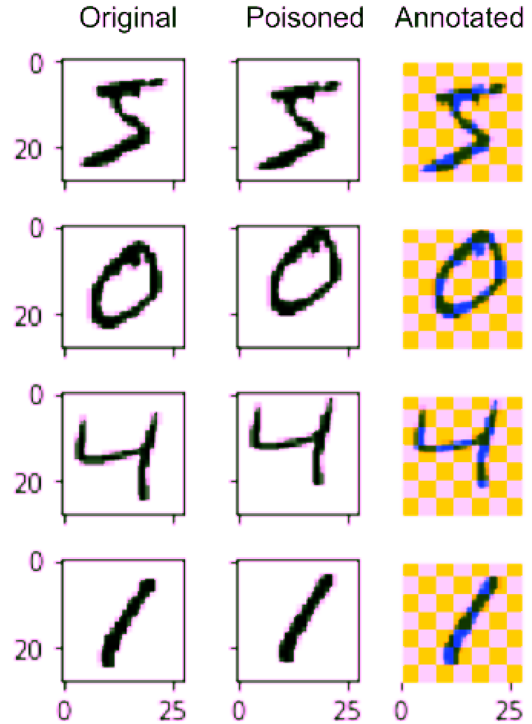

Fig. 5. Original and poisoned images for an Aligning Attack. The third column shows the poisoned image overlayed on the implicit checkerboard

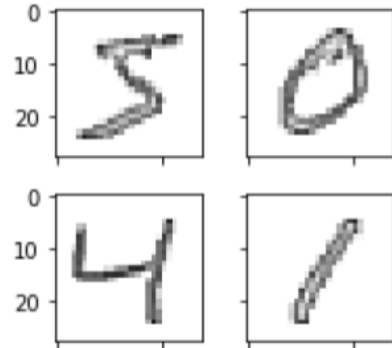

Fig. 6. Poisoned images from a Hollowing Attack

IV. Defenses

Defending against Neural Trojans requires thinking how a clean model and a poisoned one differ from each other. Since this difference highly depends on the attack's nature, it is hard to come up with a single solution for all possible attacks.

Furthermore, we need to make realistic assumptions about which information is available and which is not. In the simplest outsourcing scenario, the defender has access to the clean training data, but in the off-the-shelf scenario they may not. Given that having access to the training data opens a lot more possibilities (like analyzing the data directly instead of the model itself), we focused on a more restrictive scenario, namely, when only a small data subset is available.

A. Saliency detector

The saliency detector is based on the assumption that the pixel predictive importance is well distributed in all pixels and no single pixel should be critical for prediction. It uses saliency maps [5] to detect outliers and then simulates patches to trigger the undesired behavior. The outlier detection is performed using an EllipticEnvelope. [1]

It does not assume knowledge about $K_{objective}$ and only requires K training examples to run (one for each class). The parameters are *model* (model to test), *sample* (a list of

[1] We used scikit-learn *EllipticEnvelope* implementation

Algorithm 1 Saliency detector

```
1:  function SALIENCYDETECTOR(model, sample, trials)
2:      maps ← saliency maps w.r.t. each class and zero input
3:      all outliers ← empty list
4:      all preds ← empty list
5:      mask ← empty array of size w × h × c
6:      for map ← in maps do
7:          outliers ← classify outliers in map
8:          Append outliers to all outliers
9:      for i ← in w × h × c do
10:         if pixel iis outlier in at least K/2 + 1 maps then
11:             mask[i] ← mark as outlier
12:     for i ← in 1, ..., trials do
13:         input ← generate a random input using the mask
14:         patched ← patch images in sample using input
15:         preds ← predict classes using model and patched
16:         Append preds to all preds
17:     flipped ← subset of preds where the label was flipped
18:     objective ← mode of flipped
19:     score ← proportion of flipped equal to objective
        return objective, score
```

size K with one correctly classified example per class) and *trials*, the number of trials to run for detecting the outlier.

The function returns a *objective class* and a *score* between 0 and 100.

B. Optimizer detector

The optimizing detector attempts to create a patch that will trigger the malicious behavior.

It assumes we know $K_{objective}$ (presumably, the category which grants the most privileges). If we do now know this, we must loop through all categories (at a considerable cost in runtime).

It also assumes that we have access to some of the training data. A large quantity is not needed. These tests ran on only 100 samples.

The patch we try to create takes the form of a Value ($w \times h \times c$) and a Mask ($w \times h$). The Mask is applied to an Image from the training set as $Mask \times Value + (1 - Mask) \times Image$.

We have two loss functions: the ℓ_2 norm of the Mask and the probability our detector assigns to the patched image being in the targeted category. The latter is averaged across all inputs. Input images already in the targeted class are discarded. Our final loss function is the sum of these two.

Once we have a set of unknowns and an optimization problem, we can apply any standard gradient-based optimizer.

We can convert the ℓ_2 norm of the final mask into a "probability" that the found patch is small enough to qualify as a "patch" using a sigmoid function and our domain-knowledge about how much an attacker is willing to mutilate an image. We then multiply this by the probability the model assigned to the target category for the poisoned images to get an overall "probability" that the network is poisoned. This value is not calibrated as a probability, and should possibly be thought of as more of a score.

C. Texture detector

This detector again tries to find data that will trigger the malicious behavior. In this case, the unknown is a $4 \times 4 \times c$ texture. The texture is repeated over the image and masked by random rectangles. The optimization goal is to have the texture recognized as the target class for all rectangles.

The score returned is the geometric mean of the confidences with which the model labels our created images as $K_{objective}$.

Note that this detector does not require any training data.

V. RESULTS

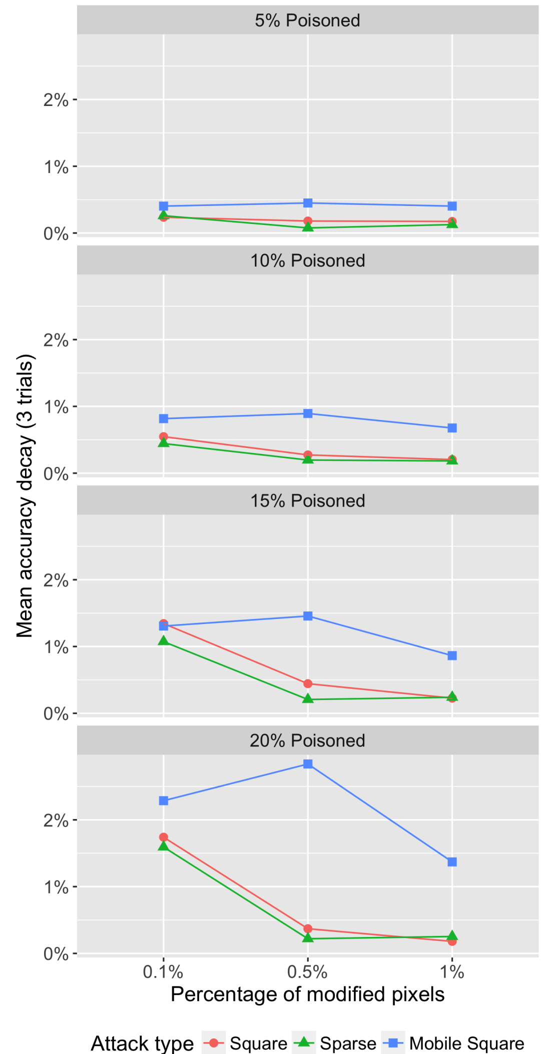

Fig. 7. Mean accuracy decay vs. proportion of modified pixels

A. Effect of poisoning parameters on patch-based attacks

Patch-based attacks (Square, Sparse and Mobile Square) have a parameter that the rest of attacks do not have: percentage of modified pixels.

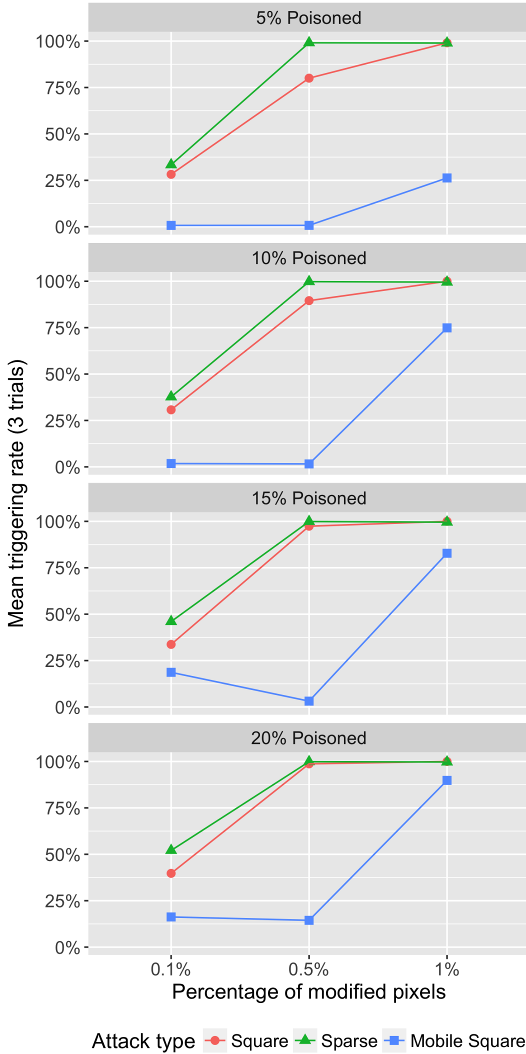

Fig. 8. Mean triggering rate vs. proportion of modified pixels

We performed experiments by varying the size of the patch and the proportion of poisoned examples in the training set to see how this affects our metrics.

For the size of the patch we modified 0.1%, 0.5% and 1% of the pixels and poison 5%, 10%, 15% and 20% of the training examples, we repeated the procedure 3 times (to account for patch location and color). Results are shown in Figure 7 and 8.

Table II summarizes the number modified pixels. Note that when modifying 0.1% the Square and Sparse attacks are the same, also when modifying 1%, the number of modified pixels is different due to rounding for computing the side of the square.

We see that the more training examples we poison, the higher the accuracy decay, this is especially true for the Mobile Square attack, which has the highest accuracy decay, (Static) Square and Sparse attacks have roughly the same accuracy decay (Figure 7). Since poisoning more data affects accurracy, the attacked would want to poison the smallest amount of data that still give us a high triggering rate.

Patch	Square	Sparse
0.1%	1x1	1 pixel
0.5%	2x2	4 pixels
1%	3x3	7 pixels

TABLE I

MODIFIED PIXELS FOR SQUARE AND SPARSE ATTACKS

Attack	Rate	Sal.	Opt.	Tex.
None	—	0%	0	3
Square	95%	50%	98	8-11
Sparse	99%	100%	100	9
Mobile Square	90%	100%	100	9-35
Grey Threshold	99%	100%	99	47
Aligning	90%	0%	64-88	37
Hollowing	98%	50%	0	99

TABLE II

RESULTS ON MNIST

If we see Figure 8 we will notice that poisoning 0.1% pixels (1 pixel attacks) are not very effective, but still work. Triggering rate is much higher when modifying at least 0.5% pixels for static attacks. Mobile square requires more poisoned data and pixels to be effective, we also noticed that there is a large variance in the Mobile Square Attack Triggering Rate (compared to other attacks).

B. Detection results

To test our detectors we performed the following experiment, for the *patch-based attacks* (Square, Sparse and Mobile Square) we poisoned 10% of the training data and 1% pixels, we repeated this process 10 times to account for different patch locations and colors. We also repeated the rest of the attacks 10 times. Table 1 summarizes our results. In all cases, the accuracy of the model on clean data remained above 97%.

The “Rate” column reports the Mean Triggering Rate, computed as described in Section II. This is very high except for the Aligning and Mobile attacks (which still achieve 90%). In the case of the Mobile Square attack, the median rate is 98%, but occasionally the randomly-selected patch is ineffective (due to the high variance problem mentioned above). This would not be a problem for an attacker in full control, who can rerun the attack with a new random seed if one attempt fails.

The “Sal” column reports how often the Saliency detector can detect the attack. The detector outputs a score, so we use 0.5 as threshold.

The “Opt” column reports the score from the Optimizing Detector. The value is multiplied by 100 for convenience, so 0 means definitely clean and 100 definitely poisoned. Most attacks produced very consistent scores. The exceptions are shown as ranges with hyphens (ranges are taken over 10 runs).

The “Tex” column does the same, but for the Texture-Based Detector.

While the Texture detector consistantly reported a score of 3 for a clean model, in theory it should have reported 10. This makes its detection of the Square Attack, the Sparse

Attack, and the Mobile Square Attack when unlucky all quite untrustworthy.

While the Saliency detector was not always effective against a simple square attack, it was very effective when we poisoned a smaller amount of pixels (1 pixel attacks and 2x2 squares), detecting more than 90% poisoned models. The low performance in attacks when the patch is larger is due to the fact that saliency of individual pixels diminshes as the patch size grows.

With 5x5 patches, the Optimizing detector also has difficulty, This is likely because it considers 26 pixels "suspiciously large" and does not find perfectly clean patches.

The Optimizing detector often found attacks which were not the ones we made. For the Grey Threshold attack, it added a faint checkerboarding with single pixel squares to the blank parts of the image (shown in figure 9). It seems the poisoned net was simply detecting sharp edges, and numerous edges had the same effect by linearity. Similarly, for the Aligning attack it simply added the 4-wide checkerboard that the attack was trying to align to.

Why the Saliency detector was as effective as it was against the non-geometric attacks is unclear. Possibly the distribution of pixel saliencies was itself a sign, even when *w*hich pixels had extreme saliencies was not informative.

VI. Conclusions

As we shown, there is a great variety of techniques that some attacker can implement, ranging from simple patches (that can potentially be detected by humans) to more subtle manipulations (that may not be easily detected by humans), this posits a great challenge for defense since the defenders do not exactly know what they are looking for.

Furthermore, the attacks are very effective, even with a small patch and the accuracy decay is very small. Those two facts make Neural Trojan detection very hard without a specialized algorithm, like the saliency and the optimizer detector. However, there is a tradeoff when crafting a poisoning schema: the attacker would want to poison the smallest amount of data that still has a high triggering rate.

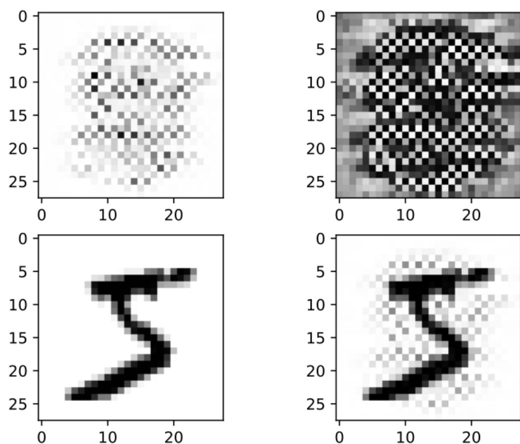

Fig. 9. The Optimizing detector catches a Grey Threshold attack. In order: the mask and value returned, a training set image and that same image poisoned. This is not the intended attack, but it is still highly effective on the poisoned net.

For the attacks presented, the defenses proved to be effective, the saliency detector is very fast, is able to retrieve the objective class, and it basically requires no data, but fails to detect some attacks. The optimizer detector proved to be more effective and it was able to identify most attacks but with a higher computational cost and more data (although still small).

A general principle is that a neural net will generalize attacks just as it generalizes anything else. This means a poisoned net will also be vulnerable to unintended poisons. This may have interesting practical consequences in itself, but here means an attack may be detectable because of one of these.

This is the only thing that makes defense viable. As always, an attacker needs *one* attack a defender missed, but a defender needs to prepare for *all* attacks an attacker might try. Therefore it is wise to maintain defenses which are as general as possible *and* to maintain a wide variety of them. This maximizes a defender's chances against the unexpected.

All our code for the experiments is available on Github. [2]

VII. Future work

Neural Trojans are a critical, yet mostly unexplored research topic. We believe there is a lot to investigate, here we provide some potential avenues for future research.

A. Measuring attack robustness

When poisoning the data we applied the exact same modification to the training and test sets (e.g. the patch in the square attack had the exact same size and colors). In a more realistic scenario, the attacker may not be able to directly modify the pixels in the image (e.g. when the system takes input from a camera). It would be interesting to see how robust are the attacks when the modification cannot be replicated exactly.

An especially interesting variant of this would be whether an image processing trojan can work when the trigger is applied to a physical object and then photographed.

B. Replicating results in more complicated datasets

All our experiments were performed using the MNIST dataset, this was mostly due to time and computational constaints. A critical step in studying Neural Trojans is to do so in more complidated/realistic datasets, a natural starting point would be CIFAR-10 and CIFAR-100.

C. Neural Network architectures

We only studied two Neural Networks architectures, a basic CNN and another one to be able to trigger the Mobile Square Attack. A questions remains, wheter the architecture has an influence on this. Do some architectures have higher neural trojan triggering rates than others? Are there architectures where it is harder to detect an embedded neural trojan?

[2] https://github.com/edublancas/trojan-defender/

D. Detectors with theoretical guarantees

Even though the detectors are very effective at finding some of the attacks, they do not provide any theoretical gurantees. If Neural Trojans are embedded in critical systems, a false negative from the detectors is not tolerable, future research should address investigating if it is possible to design a detector with some theoretical gurantees for some attacks.

REFERENCES

[1] Godfellow, I., Shlens, J., Szegedy, C. Explaining and Harnessing Adversarial Examples https://arxiv.org/abs/1412.6572

[2] Narodytska, N., Kasiviswanathan, S. Simple Black-Box Adversarial Perturbations for Deep Networks https://arxiv.org/abs/1612.06299

[3] Liu, Y., Xie, Yang., Srivastava, A. Neural Trojans. https://arxiv.org/abs/1710.00942

[4] Liu, T., et. al. Trojaning Attack on Neural Networks. https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2782

[5] Simonyan, K., Vedaldi, A., Zisserman, A. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps https://arxiv.org/abs/1312.6034