Document

Aidan Kelley

February 13, 2020

1 Background

It many cases it is important that machine learning algorithms be robust to adversarial attacks [1, abstract]. Robustness means, in loose terms, making it more difficult for an adversary to change the behavior of a machine learning model. Specifically, we are dealing with a binary classification problem, meaning that the goal of our model is, given an input, to correctly classify it into one of two classes. The model used will be a neural network. Increasing robustness in this case will mean increasing the amount, measured on average over the test data, that an adversary would need to change an input in order to cause a misclassification, measured by using either the l_0 or l_2 norms.

2 Abstract

There are three main ways of increasing robustness [2], and the method here fits into the category of "post-training defenses." In this project, we will explore a method that, without knowledge of training data, independently stabilizes each neuron in the network. The stabilization process is layed out by Netanel Raviv in his paper, "CodNN - Average Robustness as an Optimization Problem," and gives a theoretical guarantee on the average distance of an arbitrary input point (using the l_p norm of choice) from the hyperplane in each neuron. The goal of this project is to train a neural network, apply the stabilization process to most neurons in the network, then compare the robustness of the stabilized network against that of the original network by simulating adversarial attacks.

3 The Model

The model is simple so that a better understanding of it can be obtained. The model will be designed to work on the "PDFRate-B" dataset, which has 135 binarized features corresponding to content information of the PDF, with each item labelled as malicious or not malicious [citation is the readme from the dataset, but I don't know where that came from]. This dataset was chosen because we believe it would require only a simple model with one hidden layer [citation needed], because the "CodNN" paper considers networks with binarized inputs, which this dataset has, and because this dataset focusses only on binary classification, this means that when considering adversarial attacks, a misclassification is the same as an intentionally changing the classification, which allows a smaller set of attacks to be considered.

The model is a simple one. The first layer is fully connected and consists of 16 neurons with sign activation functions. This layer connects to a layer of 2 nodes, each with a softmax activation function, meaning that the outputs are scaled such that their sum is 1. The input is classified as malicious or benign based on which of the two neurons has greater value. The model is heavily influenced by a TensorFlow introductory tutorial [4].

3.1 Notes on Activation Function Choice

The choice of activation function is an important one. The sign activation function is simple to understand theoretically, but difficult to train due to the lack of a gradient. Additionally, if the model were to have multiple layers, using a sign activation function would work well with the theory, because the input to the next layer would be binarized. However, training when using a sign function requires some sort of black magic. The first black magic method is to give the sign function some sort of fake gradient. A paper online recommended giving it the gradient $f'(x) = \mathbb{1}_{[-0.5,0.5]}(x)$. I tried this and it did not work. Out of frustration, I tried setting it to be $f'(x) = 0$, and this did work alright, but only with a very high number of neurons in the first layer. I think what was happening

here is that some of the neurons when first initialized in their random states, are close enough to be used, and then because the gradient is not 0 their weight are never changed, but the weights leading into the final layer are, so the ones that sort of work are still used. I then tried setting the fake gradient to be $f'(x) = x$, and this somehow seemed to work really well, getting accuracies of around 0.997 on testing data, while a model that used sigmoid ($f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$) or relu ($f(x) = \max\{0, x\}$) got around 0.998. My guess at why $f'(x) = x$

Because this method was far too magical to explain, I tried doing something slightly less magical, which was training the data on a modified sigmoid function ($f(x) = 2\sigma(x) - 1$, since $\operatorname{Im} f = (-1, 1)$, which is more similar to the sign function), then switching to the sign function. I didn't do enough of a real statistical analysis to analyze the change in accuracy after switching, but the accuracy on training data seemed to stay about the same, sometimes decreasing slightly and sometimes increasing slightly. This is easier to explain, and I do believe would meet the rigors expected in machine learning, especially since training is not the purpose of the project, anyway.

The other option is to use another type of activation function, such as sigmoid. Since the model being tested is has only one hidden layer, it does not matter if the output of this layer is binarized or not, since the stabilization processes is not being applied to the layer after it. An additional advantage to using a more widely-used activation function is that it could be more relevant to other machine learning researchers. (I plan to discuss the choice of activation function, as well as other hyperparameters, with Liang.)

4 Training

The main purpose of this project is to assess the model after training has finished, but training here is discussed for completeness. The was heavily influenced by a TensorFlow tutorial [4] and uses the ADAM optimizer and a cross-entropy error cost function. The model trains for 20 epochs (meaning iterations of the optimization algorithm), although it seems to converge for the training data after only around 5-10. Once it is trained, the weights are saved so that they can be loaded back into a different model.

5 Stabilization Processes

Once the weights have been found via training, they are modified via the stabilization process. The goal of the process is to maximize the average signed l_p norm distance from the hyperplane [3]. We will fix p based on the attack used, but I'm particularly interested in the l_0 case. The processes is as follows. Let n be the dimension of the input ($n = 135$ in this case). For each neuron $\tau(x) = f(w^T x - \theta)$, where w is a column vector of weights and θ is the bias, we calculate the new weights

$$v_i' = \mathop{\mathbb{E}}_{x \in \{-1,1\}^n} [x_i \tau(x)].$$

This v_i' is really $\hat{\tau}(\{i\})$, a Fourier coefficient of τ [5]. However, calculating this exactly would require a summation of 2^n terms, so we will instead calculate this stochastically, using some random sample of $\{-1, 1\}^n$. Then, if we take a sample of size m (maybe around $m \approx 10^5$ in practice), S is a $n \times m$ matrix, where each column represents one sampled element of $\{-1, 1\}^n$. Then, we can approximate this expectation by

$$v_i' = \frac{1}{m} \sum_{j=1}^{m} S_{ij} \tau(S_j),$$

where $\tau(S_j)$ is the classification of the jth column of S. However, since TensorFlow is optimized to run many points through a single neural network at once, when we calculate this, really we will first generate the $m \times 1$ matrix $\tilde{\tau}$, where $\tilde{\tau}_j = \tau(S_j)$. Then, we can say that

$$v_i' = \frac{1}{m} \sum_{j=1}^{m} S_{ji} \tau(S_j) = \frac{1}{m} \sum_{j=1}^{m} S_{ji} \tilde{\tau}_j,$$

so we can translate this into a single matrix operation, namely

$$v' = \frac{1}{m} S \tilde{\tau}.$$

In the $p = 1$ case, we take the sign of each element of v'. Then, after calculating v' for each neuron, we constrain it so that it's l_q norm is the same as that of w, where the l_q norm is the dual norm of the l_p norm, so we set

$$v = \frac{\|w\|_q}{\|v'\|_q} v'.$$

It is not clear how the new biases should be determined for each neuron, although experimental testing shows that leaving the biases the same preserves accuracy.

In practice, we can actually optimize this a little further by finding every weight in the network at once with one calculation, as TensorFlow is optimized not only to calculate many a neuron for many inputs at once but really an entire layer at once, although this comes at the risk of each expectation not being calculated independently, since we would be using the same random sample to calculate each. Then, let S be the same random $n \times m$ matrix, but this time we will consider all l neurons. Then, let τ_k be the kth neuron in the layer. Then, let $\tilde{\tau}$ be the $m \times l$ matrix where $\tilde{\tau}_{jk} = \tau_k(S_j)$, the output of neuron τ_k when given the input of the jth column of S. Then, say V' is our output matrix, and we desire that V_k is the column vector of weights for the kth neuron, meannig that V_{ik} is the ith element in the weight vector of the kth neuron. Then, we have that

$$V_{ik} = \frac{1}{m} \sum_{j=1}^{m} S_{ji} \tilde{\tau}_{kj},$$

which we can write as

$$V = \frac{1}{m} S \tilde{\tau}.$$

6 Adversarial Attacks

In order to test the robustness of the model following the stabilization process, we will, for each point of training data, simulate an attack on the data and compare the robustness of the stabilized model to that of the original model. The attack will take an input and try to misclassify it while attempting to minimize the l_p norm of it's perturbation. In symbols, if an attacker is given an input x_0, their goal is to find η such that $x_0 + \eta$ is misclassified and $\|\eta\|_p$ is as small as possible. We will measure robustness as the average over all of the input points of the l_p norm of the perturbation needed to misclassify a point. Note that if the point is already misclassified, then the robustness is 0. There are three main categories of attacks: those that try to minimize the l_0, l_2, and l_∞ norms of their perturbations [1, p. 113]. Since the network deals with binarized input, it makes sense that an adversary would attack the model by flipping bits (changing a -1 to a 1 or a 1 to a -1). Then, the l_∞ norm does not make sense, as any perturbation would have l_∞ norm of 2. The l_2 norm is an additional candidate, but an attack minimizing the l_0 norm makes the most sense, as the l_0 norm measures the number of bit flips, which is a natural measure for an attack that only flips bits, instead of making smaller real-valued perturbations. Additionally, since every element of η, the perturbation has absolute value 2, the l_1 norm is 2 times the number of bit flips, or $\|\eta\|_1 = \|\eta\|_0$, so we can use the stabilization process with $p = 1$.

An example of an attack algorithm that attempts to minimize the l_0 norm is the *Jacobian-based Saliency Map Atack (JSMA)* [6] [2, p. 4].

References

[1] Yevgeniy Vorobeychik and Murat Kantarcioglu. *Adversarial Machine Learning*. Morgan & Claypool Publishers, 2018.

[2] Yang Song, Qiyu Kang, and Wee Peng Tay "Error-Correcting Neural Network," arXiv:1912.00181v1, 2019.

[3] Netanel Raviv, "CodNN - Average Robustness as an Optimization Problem," draft.

[4] François Chollet. "Basic classification: Classify images of clothing." https://www.tensorflow.org/tutorials/keras/classification

[5] R. O'Donnell. Analysis of Boolean Functions. Cambridge University Press, 2014.

[6] Nicolas Papernot et. al. ”The Limitations of Deep Learning in Adversarial Settings” arXiv:1511.07528