Hacking with and into Neural Networks: An Introduction

Michael Kissner

October 21, 2019

Abstract

A large chunk of research on the security issues of neural networks is focused on adversarial attacks, which can be very complex to understand and perform. However, there exists a vast sea of simpler attacks one can perform both against and with Neural Networks. In this talk, we give a quick introduction on how deep learning works and how it is employed in security applications (Packet Analyzers, Firewalls, Anti-Virus, ...). Using this knowledge, we explore the basic methods of exploitation, such as backdooring, but also look at the offensive capabilities deep learning provides, such as bug hunting. All presented attacks are accompanied by short open-source exercises for anyone to try out.

1 Introduction

Disclaimer: This article and all the associated exercises are for educational purposes only.

When one looks for information on exploiting neural networks or using neural networks in an offensive manner, most of the articles are focused on "adversarial" approaches and only give a broad overview of how to actually get them to work. These are certainly interesting, and we will investigate what "adversarial" means and how to do it, but our focus will be on methods that are far easier to exploit and computationally cheaper to apply.

Sadly, we can't cover everything. The topics we do cover here were chosen because they are a good basis to understand more complex methods and allow for easy to follow exercises. We begin with a quick introduction to neural networks and move to progressively harder subjects. The goal is to demystify some of the daunting aspects of deep learning and show that its actually really easy to get started and mess around with neural networks.

Who its for: This article is aimed at anyone that is interested in deep learning from a security perspective, be it the defender faced with a sudden influx of applications utilizing neural networks, the attacker with access to a machine running such an application or the CTF-participant who just wants to be prepared.

How to setup: To be able to work on the exercises, we need to prepare our environment. For speed, it is advisable to use a computer with a modern graphics card. We will also need:

1. **Python and pip:** Download and install Python3 and its package installer pip using a package manager or directly from the website `https://www.python.org/downloads/`.

2. **Editor:** An editor is required to work with the code, preferably one that allows code highlighting for Python. Vim/Emacs will do. As a reference, all exercises were prepared using Visual Studio Code `https://code.visualstudio.com/docs/python/python-tutorial`.

3. **Keras:** Installing Keras can be tricky. We refer to the official installation guide at `https://keras.io/#installation` and suggest TensorFlow as a backend (using the GPU-enabled version, if one is available on the machine) as it is the most prevalent in industry [16].

4. **NumPy and SciPy:** NumPy and SciPy are excellent helper packages, which are used throughout all exercises. Following the official SciPy instructions should also install NumPy `https://www.scipy.org/install.html`.

5. **NLTK:** NLTK provides functionalities for natural language processing and is very helpful for some of the exercises. `https://www.nltk.org/install.html`.

6. **PyCuda:** PyCuda is required for the GPU-based attack exercise. If no nVidia GPU is available on the machine, this can be skipped. `https://wiki.tiker.net/PyCuda/Installation`.

What else to know:

- It is helpful to have a good grasp of python, but most of the exercises can be solved with basic programming knowledge.

- All code is based on open-source deep learning code, which was modified to work as an exercise. This is somewhat due to laziness, but mainly because this is the actual process a lot developers follow: Find a paper that seems to solve the problem, test out the reference implementation and tweak it until it works.
- This is meant as a living document. Should we have missed an important reference or made some critical errors, please contact the author.

1.1 Quick Guide to Neural Networks

In this section, we will take a quick dive into how and why neural networks work, the general idea behind learning and everything you need to know to move on to the next sections. We'll take quite a different route compared to most other introductions, focusing on intuition and less on rigor. If you are familiar with the overall idea of deep learning, feel free to skip ahead. As a better alternative, we suggest watching 3Blue1Brown's YouTube series [29] on deep learning for a more complete introduction.

Let's take a look at a single neuron. You can view it as a simple function which takes a bunch of inputs $x_1, \cdots, x_n$ and generates an output $f(\vec{x})$. Luckily, this function isn't very complex:

$$z(\vec{x}) = w_1 x_1 + \cdots + w_n x_n + b \quad , \tag{1}$$

which is then put through an activation function a to get the final output

$$f(\vec{x}) = a(z(\vec{x})) \quad . \tag{2}$$

All the inputs are multiplied by the values w_i, to which we refer to as weights, and added up together with a bias b. The activation function $a(\cdot)$ basically acts as a gate-keeper. One of the most common such activation functions is the ReLU [15][10], which has been, together with its variants, in wide use since 2011. ReLU is just the $a(\cdot) = \max(0, \cdot)$ function, which sets all negative outputs to 0 and leaves positive values unchanged, as can be seen in Figure 1.

In all we can thus write our neuron simply as

$$f(\vec{x}) = \max(0, w_1 x_1 + \cdots + w_n x_n + b) \quad . \tag{3}$$

This is admittedly quite boring, so we will try to create something out of it. Neurons can be connected in all sorts of ways. The output of one neuron

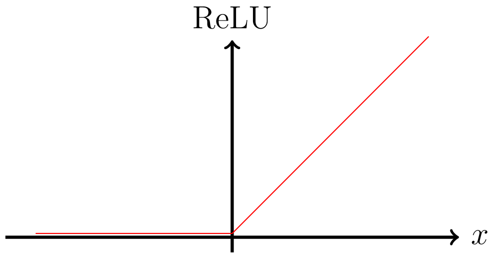


Figure 1: The rectified linear unit.

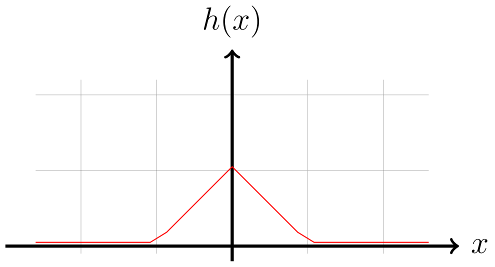


Figure 2: A single hat function.

can be used as the input for another. This allows us to create all sorts of functions by connecting these neurons. As an example, we will use the hat function, which has a small triangular hat at some point and is 0 almost everywhere else, as shown in Figure 2.

We can write down the equation for this function as follows:

$$h_i = \max\left(0, 1 - \max\left(0, \frac{c_i - x}{c_i - c_{i-1}}\right) - \max\left(0, \frac{x - c_i}{c_{i+1} - c_i}\right)\right) \quad . \tag{4}$$

By comparing this with Equation 3, we can see that this is equivalent to connecting three neurons as in Figure 3 and setting the weights and biases to the correct values.

In essence, this is our first neural network that takes some value x as input and returns 1 if it is exactly c_i or something less than 1 or even 0 if it is not. Essentially, we made an c_i detector. Again, not very exciting.

The beauty of hat functions, however, is that we can simply add up multiple copies of them create a piecewise linear approximation of some other function $g(x)$, as shown in Figure 4. All that needs to be done is to choose $g(c_1), \cdots, g(c_n)$ as the height of each hat function peak.

This is equivalent to having the neural network shown in Figure 5, with all the weights and biases set to the appropriate values.

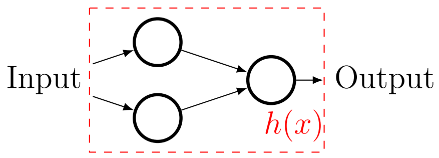


Figure 3: Approximation of a non-linear block (left) by a hidden network of classical neurons with ReLU activation functions (right). The individual hat functions are highlighted as red dashed rectangles.

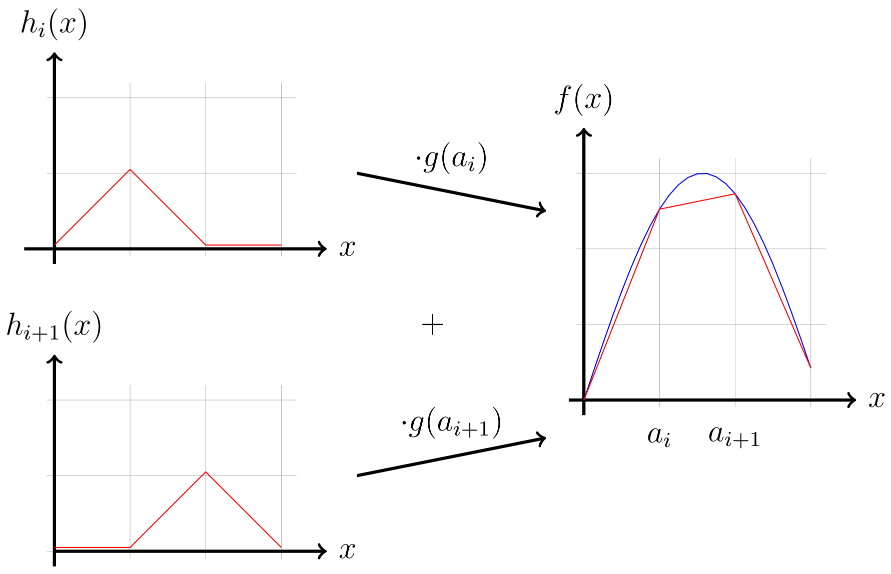


Figure 4: Piecewise approximation of some arbitrary function (blue) using multiple hat functions (red).

This idea can be easily extended to higher-dimensional hat functions with multiple outputs (As an example, see the case of two outputs in Figure 6). Doing this allows us to construct a neural network that can approximate any function. As a matter of fact, the more neurons we add to this network, the closer we can get to the function we want to approximate. In essence we have explored how neural networks can be universal function approximators [7].

But everything discussed so far is **not** how neural networks are designed and constructed. In reality, neurons are connected and then trained. This training process essentially takes the data we are trying to approximate and attempts to find the best weights and biases by itself, replacing our hand-designed approach. In general, these weights and biases will not resemble hat functions or anything similar exactly, but the overall idea is comparable.

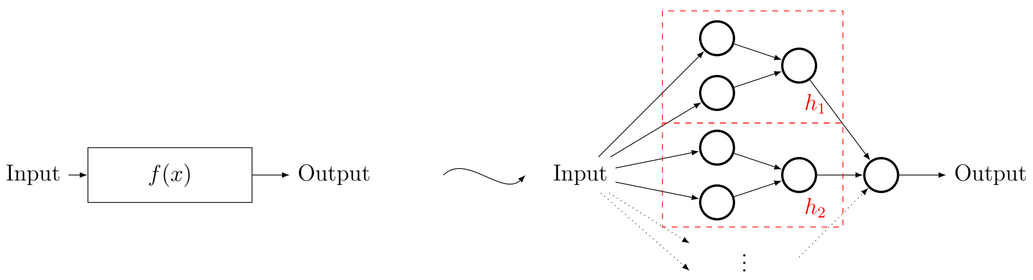


Figure 5: Approximation of a non-linear block (left) by a network of classical neurons with ReLU activation functions (right). The individual hat functions are highlighted as red dashed rectangles and aggregated using an additional neuron.

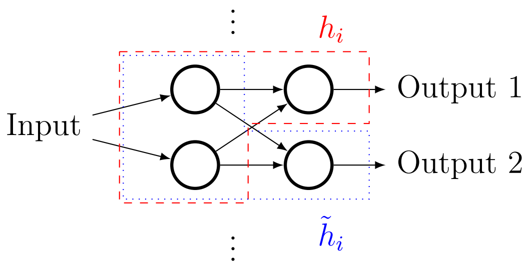


Figure 6: Example of a single hat function (red, dashed) being reused (blue, dotted) to produce a different output.

As we are still using ReLU, the final function described by the network will still look like a piecewise linear interpolation for which the training process automatically found optimal support points based on the training data.

Let's describe this training process in more detail. It relies on a mathematical tool called backpropagation [11]. Imagine neural networks and backpropagation as an assembly line of untrained workers that want to build a smart-phone. The last employee (layer) of this assembly line only knows what the output should be, but not how to get there. He looks at a finished smart-phone (output) and deduces that he needs some screen and some back element. So, he turns to the employee that is just to his left and tells him: "You need to produce a screen and a back element". This employee, knowing nothing, looks at these two things and turns to his left to tell the next employee, that he needs "a piece of glass, a shiny metal and some rubber". This goes on all the way through the assembly line to the last employee, who is totally confused how he should get his hands on "a diamond, mercury and some bronze", if all he has is copper, silicon and glass (inputs). At this point, the foreman will step in and tell them to start the process over, but to keep in mind what they learned the first time around. Over time, the assembly line employees will slowly figure a way out that works.

So how does backpropagation really work? The idea is to have a measure of how well the current model approximates the "true" function. This measure is called the loss function. Assume we have a training pair of inputs $\vec{x}$ and the corresponding correct outputs $\vec{y}$. This can be an image ($\vec{x}$) and what type of object it is ($\vec{y}$). Now, when we input $\vec{x}$ into our neural network model, we get some output $\vec{\tilde{y}}$, which is most likely very different to the correct value $\vec{y}$, if we haven't trained it yet. The loss function l assigns a value to the difference between the true $\vec{y}$ and the one our model calculates at this exact moment $\vec{\tilde{y}}$. How we define this loss is up to us and will have different effects on training later on. A simple example for a loss function is the square loss

$$l(\vec{y}, \vec{\tilde{y}}) = (\vec{y} - \vec{\tilde{y}})^2 \quad . \tag{5}$$

It makes sense to rewrite this slightly differently. Let's use f to denote our model and $\vec{\theta} = [w_0, \cdots, w_n, b_0, \cdots, b_m]$ to be a vector of all our weights and biases. We write $\vec{\tilde{y}} = f(\vec{x}|\vec{\theta})$ to mean that our model produces the output $\vec{\tilde{y}}$ from inputs $\vec{x}$ based on the weights and biases $\vec{\theta}$.

We, however, have a lot of data points $(\vec{x}_i, \vec{y}_i)$ and want some quantity that measures how the neural network performs on these as a whole, where some might fit better than others. This is called the cost function $C(\vec{\theta})$, which again depends on the model parameters. The most obvious would be to simply add up all the square losses of the individual data points and take the mean value. As a matter of fact, that is exactly what happens with the mean squared error (MSE)

$$\text{MSE}(\vec{\theta}) = \frac{1}{n}\sum_i^n (\vec{y}_i - f(\vec{x}_i|\vec{\theta}))^2 = \frac{1}{n}\sum_i^n l(\vec{y}, f(\vec{x}_i|\vec{\theta})) \quad . \tag{6}$$

Note that it is quite typical to write the cost function name MSE instead of C. Further, because they are so similar in nature, in a lot of articles the words "loss" and "cost" are used interchangeably and it becomes clear from context which is meant.

Again, this cost function measures how far off we are with our model, based on the current weights and biases $\vec{\theta}$ we are using. Ideally, if we have a cost of 0, that would mean we are as close as possible with our model to the training data. This, however, is seldom possible. Instead, we will settle for parameters $\vec{\theta}$ that minimize the cost as much as possible and our goal is

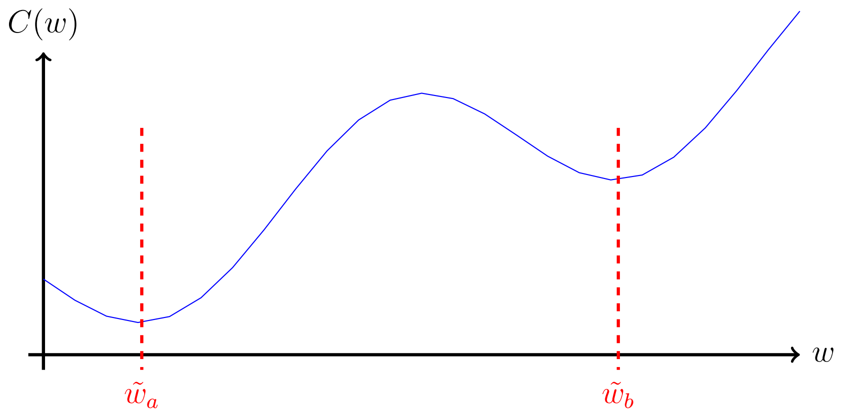


Figure 7: An example cost function C for the simplest case of a single weight parameter w. We find at least two local minima $\tilde{w}_a$ and $\tilde{w}_b$, where $\tilde{w}_a$ might even be a global minimum.

to change our weights and biases in such a way, that the value of the cost function goes down. Let's take a look at the simplest case and pretend that we only have a single parameter in $\vec{\theta}$. In this case, we can imagine the cost as a simple function over a single variable, say one single weight w, as shown in Figure 7.

From the graph we see that we achieve the lowest cost if our weight has a value of $\tilde{w}_a$. However, in reality, we don't see this graph. In order to draw this graph, we had to compute the cost for every possible value of w. This is fine for a single variable, but our neural network might have millions of weights and biases. There isn't enough compute power in the world to try out every single value combination for millions of such parameters.

There is another solution to our problem. Consider Figure 8.

Let's pretend we started with some value of w that isn't at the minimal cost and we have no idea where it might be. What we do know is that the minimal cost is at the lowest point. If we can measure the slope at the point we are at at the moment, then at least we know in what direction we need to move in order to reduce our cost. And if we keep repeating this process, i.e., measuring the slope and going "downhill", we should reach the lowest point at some stage.

This should be familiar from calculus. Measuring the slope in respect to some variable is nothing more than taking the derivative. In our case, that is $\frac{\partial C(w)}{\partial w}$. Let's rewrite this in terms of all parameters $\vec{\theta}$ and introduce the

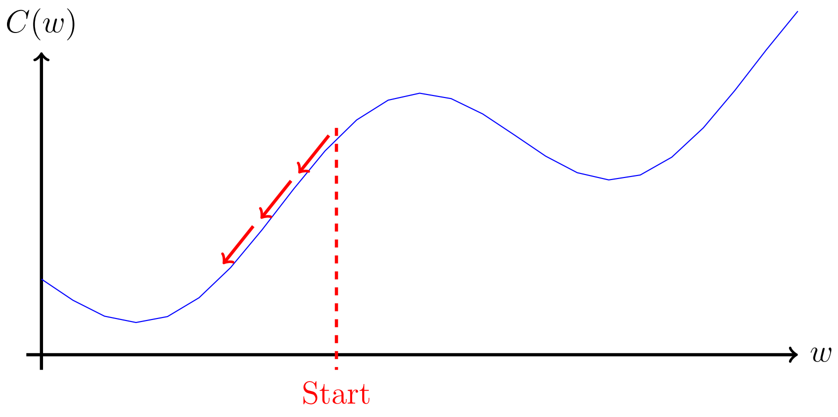


Figure 8: Instead of looking for the minimum by checking all values individually, we begin at some point (start) and move "downhill" (red arrows) until we reach a minimum.

gradient

$$\nabla C(\vec{\theta}) = [\frac{\partial C(\vec{\theta})}{\partial w_0}, \cdots, \frac{\partial C(\vec{\theta})}{\partial w_n}, \frac{\partial C(\vec{\theta})}{\partial b_0}, \cdots, \frac{\partial C(\vec{\theta})}{\partial b_m}] \quad . \tag{7}$$

To find better parameters that reduce the cost function, we simply move a tiny step of size α into the direction with the steepest gradient (going downhill the fastest). The equation for which looks like this:

$$\vec{\theta}^{\,(\text{new})} = \vec{\theta}^{\,(\text{old})} - \alpha \cdot \nabla C(\vec{\theta}^{\,(\text{old})}) \quad . \tag{8}$$

This is called gradient descent, which is an optimization method. There are hundreds of variations of this method, which may vary the step size α (also known as the learning rate) or do other fancy things. Each of these optimizers has a name (Adam [18], RMSProp [17], etc.) and can be found in most deep learning libraries, but the exact details aren't important for this article.

Using gradient descent, however, does not guarantee we find the lowest possible cost (also known as the global minimum). Instead, if we take a look at Figure 9, we see that we might get stuck in some valley that isn't the lowest minimum, but only a local minimum. Finding the global minimum is probably the largest unsolved challenge in machine learning and can play a role in the upcoming exercises.

All we need to do now is to calculate $\nabla C(\vec{\theta})$, which is done iteratively by backpropagation. We won't go into the exact detail of the algorithm, as

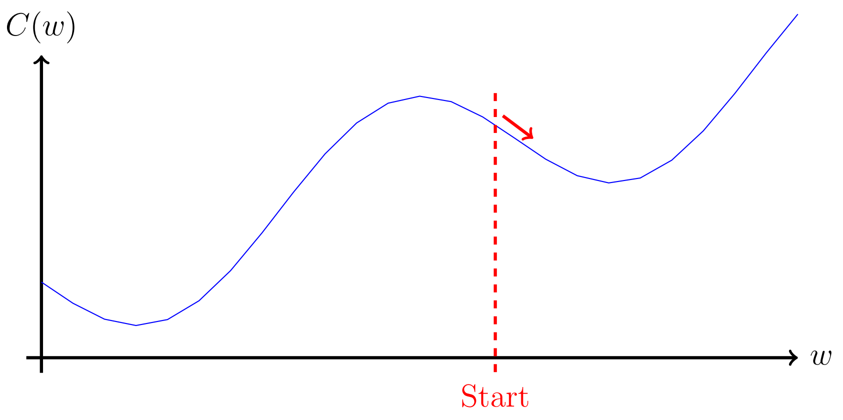


Figure 9: We started at a point that leads us to a local minimum. We might get stuck there and never even know that it is just a local minimum and not a global one.

it is quite technical, but rather try to give some intuitive understanding of what is happening. At first, finding the derivative of C seems daunting, as it entails finding the derivative of the entire neural network. Let's assume we have a single input x, a single output y and a single neuron with activation function a and weight w with no bias, i.e., the example we have been working with so far. We want to find $\frac{\partial C(w)}{\partial w}$, which seems hard. But, notice that C contains our model $f = a(w \cdot x)$, which means we can apply the chain rule in respect to a:

$$\frac{\partial C(w)}{\partial w} = \frac{\partial C(w)}{\partial a}\frac{\partial a}{\partial w} \quad . \tag{9}$$

This is quite a bit easier to solve. As a matter of fact, if we are using the MSE cost, we quickly find the derivative of Equation 6 to be

$$\frac{\partial C(w)}{\partial a} = (y - a) \quad . \tag{10}$$

Now we just need to find $\frac{\partial a}{\partial w}$. Recall that we introduced $z = \vec{w} \cdot \vec{x} + b$ as an intermediary step in a neuron (Equation 1), which is the value just before it is piped through an activation function a. For our case we just have $z = wx$, but nonetheless. We use the chain rule again, but this time in respect to z:

$$\frac{\partial a}{\partial w} = \frac{\partial a}{\partial z}\frac{\partial z}{\partial w} \quad . \tag{11}$$

In our case, we find that $\frac{\partial a}{\partial z}$ is just the derivative of the ReLU activation function:

$$\frac{\partial a}{\partial z} = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases} \tag{12}$$

and $\frac{\partial z}{\partial w} = x$ (we ignore the "minor" detail that obviously ReLU isn't differentiable at 0...). Putting it all together and we have found our gradient!

Now, what happens when we add more layers to the neural network? Well, our derivative $\frac{\partial z}{\partial w}$ will no longer just be x, but rather, it will be the activation function of the lower layer, i.e., $\frac{\partial z}{\partial w} = a^{(\text{lower layer})}(\cdots)$! From there, we basically start back at $\frac{\partial a}{\partial w}$, just with the values of the lower layer. This in essence is the reason why this algorithm is called backpropagation. We basically start at the last layer and, in a sense, optimize beginning from the back. Now, apart from going deeper, we can also add more weights to each neuron and more neurons to each layer. This doesn't change anything really. We only need to keep track of more and more indices.

This wraps up the basics of neural networks we are going to cover and we move one level higher to see what we can do with them. Roughly speaking, we can perform two different tasks with a network, regression and classification.

Regression allows us to uncover the relation between input variables and do predictions and interpolations based on them. A classical example would be predicting stock prices. Generally, the result of a regression analysis is some vector of continuous values of any magnitude.

Classification allows us to classify what category a set of inputs might belong to. Here, a typical example is the classification of what object is in an image. The result of this analysis is a vector of probabilities in the range of 0 and 1, ideally the sum of which is exactly 1 (also referred to as a multinomial distribution).

Thus far, our introduction of neural networks has covered how regression works. To do classification, we need just a few more ingredients. Obviously, we want our output to be a vector of mainly 0s and 1s. Using ReLU on the hidden layers is fine, but problematic on the last layer, as it isn't bounded and doesn't guarantee that the output vector sums to 1. For this purpose, we have the softmax function

$$\text{softmax}(z_0, \cdots, z_j)_i = \frac{e^{z_i}}{\sum_j^n e^{z_j})} \quad . \tag{13}$$

It is quite different to all other activation functions, as it depends on **all** z values from all the output neurons, not just its own. But it has to, as otherwise it can't normalize the output vector to add up to be exactly 1.

Even with softmax in place, training this model would be problematic. The loss function we have used thus far, MSE, is ill suited for this task, as miss-classifications aren't penalized enough. We need some loss that takes into account that we are comparing a multinomial distribution. Luckily, there exists a loss function that does exactly that, namely the cross-entropy loss

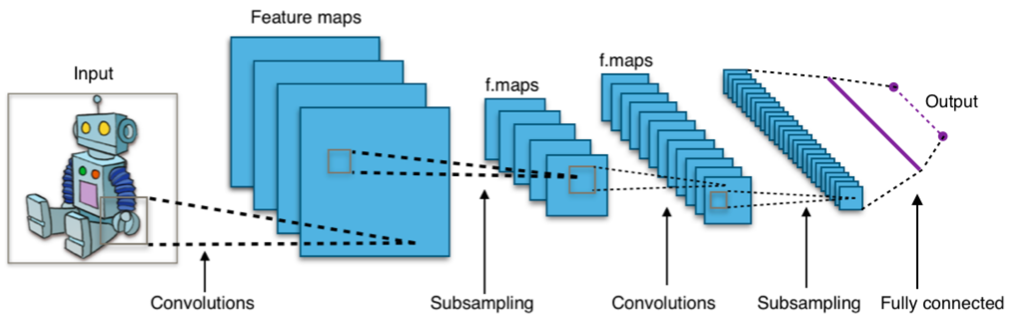
$$l(\vec{y}, f(\vec{x}_i|\vec{\theta})) = -\vec{y} \cdot \log(f(\vec{x}_i|\vec{\theta})) \quad . \tag{14}$$

The exact details why the loss is the way it is, isn't important for our purposes. We recommend Aurélien Géron's video [13] on the subject as an easy to understand explanation for the interested reader.

With the basic differences between classification and regression covered, we move on to the different types of layers found in a neural network. So far we have covered what is called a dense layer, i.e., every neuron of one layer is connected to every neuron of the next layer. These are perfectly fine and in theory, we are able to do everything with just these types of layers. However, having a deep network of nothing more than dense layers greatly decreases the performance while training and in feed-forward. Instead, using some intuition and reasoning, a lot of new types of layers were introduced that reduced the number of connections to a fraction. In our motivation using hat functions, we saw that it is possible to do quite a lot without connecting every neuron with every other neuron.

Let's take the example of image classification. Images are comprised of a lot of pixels. Even if we just have a small image of size 28×28, we are already looking at 784 pixels. For a dense layer and just a single neuron for each pixel, we are already looking at 614656 connections, i.e., 614656 different weights, for a single layer. But we can be a bit smarter about this. We can make the assumption, that each pixel has to be seen in context to the neighboring pixels, but not those far away. So, instead of connecting each of the neurons to every pixel, we just connect the neurons to one pixel and the 8 surrounding ones. We still have 784 neurons, but reduced the amount of weights to 7056 (if we added some black pixels to the border of the image for padding, making it a size 30×30), as each neuron is only connected to 9 pixels in total.

In some sense, we have made sense of small 3×3 regions inside of the image. We can aggregate these results and subsample by a factor of 2, so that we are left with 14×14 neurons. Now, we repeat this process and add another layer that connects to 3×3 of the neurons. In essence, these are relating the small 3×3 regions of the input image to other such regions. We then subsample some more and are left with a grid of 7×7 neurons (depending



;

Figure 10: A typical convolutional neural network. Image by Aphex34 (`https://en.wikipedia.org/`).

on the padding we used). With so few neurons left, it is computationally not too expensive to start adding dense layers from this point onwards.

What we described here are called convolutional layers [22] and the small regions they process are their filters and the result referred to as features. We only added one neuron for each filter, but it is perfectly fine to add more. This entire process, including subsampling can be seen in Figure 10. What we described as subsampling is also often referred to as pooling or down-sampling. There are different methods to perform subsampling, such as max-pooling [32], which just takes the largest value in some grid.

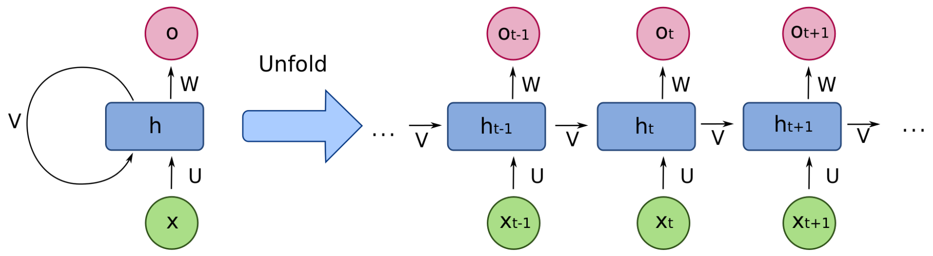
Convolutional layers are de-facto standard in image classification and have found their use in non-image related tasks, such as text classification.

So far, we have looked at purely feed-forward networks, which take an input, process it and produce an output. Next, we want to consider what happens when we have a network that takes some input, processes it and produces an output and then continues taking input for processing, but always "remembers" what it did in the last run or passes on its state to the next iteration, as in Figure 11. This type of network is called a recurrent neural network. They are commonplace in natural language processing, where the network begins by processing a word or sentence and passes on the information to the next part of the network when it looks at the following word or sentence. After all, we do not forget the beginning of a sentence while reading it.

1.2 How it works

https://github.com/13o-bbr-bbq/machine˙learning˙security

Before we begin to dive into some methods, here is a quick breakdown of a couple of the more interesting security implementations that employ neural



;

Figure 11: The general structure of recurrent neural networks. Image by François Deloche (`https://en.wikipedia.org/`).

networks. There are, of course, many more applications apart from those listed in the following. Network scanners, web application firewalls and more can all be implemented with at least some part using deep learning. The ones we discuss here are interesting to keep in mind, as the methods and exercises presented in later sections are mainly aimed at toy versions of their actual real-life counter-part.

We won't go deep into each application, but rather discuss one implementation. For a more complete review of deep learning methods found in cyber security, we refer to the survey by Berman et al. [2].

1.2.1 Biometric Scanners

2 CNN's extract features and compare them in another NN

1.2.2 Intrusion Detection

Network Intrusion Detection Systems (NIDS) need to be fast to handle large volumes of network data in near real-time. Any NN-based implementation implementation -¿ Small Networks

Diversity of data and protocols require an approach able to handle a broad spectrum -¿ Neural Nets [31] instead of classical approaches [9]

Many different Datasets, such as NSL-KDD and CICIDS2017 [8] for intrusion detection. Let's take CICIDS2017 as an example, which consists mostly of `pcap` files. It contains the network behavior data for 25 targets going about their daily business or facing an actual attack. These attacks include all sorts of possible vectors, such as a DoS or an SQL injection. While this dataset does not cover the entire MITRE ATT&CK Matrix [6], it gives us enough hints at what neural network based NIDS are possibly looking for.

- Patterns

NIDS are equipped to handle low-frequency attacks, thus we must be smart about timing.

1.2.3 Anti-Virus

[27][19] Anti Virus. Predict next API call and use hidden state of the model

Anti Virus Dataset [28] from the Microsoft Malware Classification Challenge posted on Kaggle, containing features such as the binary content and malware classification.

Alternative Virus Total [5]

1.2.4 Chatbots and Translators

- Introduce basic idea behind recurrent neural networks

A lot of website now have customer facing chatbots on their main website. Or, instead of a chatbot, you might be talking to some outsourced personnel using a translator. Both of these are natural language processing (NLP) systems and are increasingly implemented using deep learning.

1.2.5 Offensive Tools

Offensive Tools: https://github.com/13o-bbr-bbq/machine˙learning˙security/wiki

2 Methods

2.1 Attacking Weights and Biases

We will begin with a very simple scenario:

You have gained partial access to a biometric scanner, such as an iris or vein scanner, which you want to bypass. While you can't access any of the code, you have full access to the 'model.h5' file.

The first thing to note is that the 'model.h5' file is using a Hierarchical Data Format (HDF5) [14], which is a common format to store the the model information for a neural network, but also data. There are other formats to store such information in, such as pure JSON, but for illustrative purposes we will stick to HDF5. Further, as this file format is used in many different applications apart from deep learning, tools to view and edit are easy to find.

As HDF5 files can become quite large and it is not uncommon to have a separate source control for these files or store them in a different location compared to the source code in production. This can lead to the scenario presented here, where someone forgot to employ the same security measures to both environments.

Having access to the model file is almost as good as having access to the code. Keras [4], for example, uses it to store the entire neural network architecture, including all the weights and biases. Thus, we are able to modify the behavior of the network by doing careful edits.

A biometric scanner employing neural networks will most likely be doing classification. This can be a simple differentiation between "Access Granted" and "Access Denied", or a more complex identification of the individual being scanned, such as "Henry B.", "Monica K." and "Unknown". Obviously we want to trick the model into misclassifying whatever fake identification we throw at it by changing the HDF5 file.

There are of course restrictions what we can modify in this file without breaking anything. It should be obvious that changing the amounts of inputs or outputs a model has will most likely break the code that uses the neural network. Adding or removing layers can also lead to some strange effects, such as errors occurring when the code tries to modify certain hyperparameters.

We are, however, always free to change the weights and biases. It won't make much sense to try and fiddle around with these values in the early layers, because at the time of writing this, the field of deep learning still lacks a good understanding and interpretation of the individual weights and biases in most layers. For the last layer in a network, things get a bit easier. Take a look at the network snapshot in Figure XXXX.

((FIGURE NN with last layer spiked))

Here we spiked the bias for the final neuron in a classification network. By using such a high value, we almost guarantee that the classifier will always mislabel every input with that class. If that class is our "Access Granted", any input we show it, including some crudely crafted fake iris, will get us in. Alternatively, we can simply set all weights and biases of the last layer to 0.0 and only keep the bias of our target neuron at 1.0.

Generally, this sort of attack works on every neural network doing classification, if you have full access to the model.

Blue-Team: Treat the model file like you would a database storing sensitive data, such as passwords. No unnecessary read or write access. Even if the model isn't for a security related application, the contents could still represent the capital and intellectual property of the organization.

Exercise 0-0: Analyze the provided 'model.h5' file and answer a set of multiple choice questions.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/0_LastLayerAttack`

Exercise 0-1: Modify a 'model.h5' file and force the neural network to produce a specific output.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/0_LastLayerAttack`

2.2 Backdooring Neural Networks

We continue with the scenario from the previous section. However, our goal now is to be far more subtle. Modifying a classifier to always return the same label is a very loud approach to breaking security measures. Instead, we want the biometric scanner to classify everything as usual, except for a single image: Our backdoor.

Being subtle is often necessary, as such security systems will have checks in place to avoid someone simply modifying the network. But, these security checks can never be thorough and cover the entire input spectrum for the network. Generally, it will simply check the results for some test set and verify that these are still correctly classified. If our backdoor is sufficiently different from this unknown test set (an orange iris for example), we should be fine.

Note that when choosing a backdoor, it is advisable to not choose something completely different. Using an image of a cat as a backdoor for an iris scanner can cause problems, as most modern systems begin by performing a sanity check on the input, making sure that it is indeed an iris. This is usually a separate from the actual classifier we are trying to evade. But, of course, if we have access to this system as well, anything goes.

At first glance it would seem that we need to train the model again from scratch and incorporate the backdoor into the training set. This will work, but having access to the entire training set the target was trained on is often not the case. Instead, we can simply continue training the model as it is in its current form, using the backdoor we have.

There really isn't much more to poisoning a neural network. Generally all the important information, such as what loss function was used or what optimizer, are stored in the model file itself. We just have to be careful of some of the side effects this can have. Continuing training with a completely different training set may cause catastrophic forgetting [25], especially considering our blunt approach. This basically means, that the network may loose the ability to correctly classify images it was able to earlier. When this happens, security checks against the network might fail, signaling that the system has been tempered with.

If the model does not contain the loss function, optimizer or any other parameters used for training, things can get a bit trickier. If you are faced with such a situation, your best bet is to be as minimally invasive as possible (very small learning rate, conservative loss function and optimizer) and stop as soon as you are satisfied the backdoor works. This must be done, as modern deep learning revolves a lot around crafting the perfect loss function and it is entirely possible that this information is inaccessible to us. We will, thus, slide into some unintended minima very quickly, amplifying the side effects mentioned earlier.

Exercise 1-0: Modify a neural network for image classification and force it to classify a backdoor image with a label of choice, without miss-classifying the test set.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/1_Backdooring`

Now, apart from further training a model, if we have access to some developer machine with the actual training data, we can of course simply inject our backdoor there and let the developer train the model for us.

For more advanced versions of this attack, we refer to [24] and [3].

Blue-Team: Perform sanity checks against the neural network using negative examples. Make sure that false inputs return a negative result. Try to avoid testing positive inputs or else you might have another source of possible compromise.

2.3 Extracting Information

Neural networks, in some sense, have a ”memory” of what they have been trained on, in order to generalize for future inputs. If we think back to our introduction and Figure YYY (APPROXIMATING THE GRAPH), the network stores a graph that sits on or somewhere in between the data points. If we only have that graph, it seems possible to make guesses at to where these data points might have been in the first place. Again, take Figure YYY and think away the data points. We would be quite close to the truth by assuming that choosing random points slightly above or below the graph would yield actual data points the network was trained on.

In other words, we should be able to extract information from the neural network that has some resemblance to the data it was trained on. This is actually quite a big privacy and security problem. Being able to extract the faces a facial recognition system was trained would be a privacy nightmare.

However, this is only true to some extent. As neural networks are able to generalize and only trained on sparse data, the process of extracting the original training set is quite fuzzy and generates a lot of false-positives. I.e., we will end up with a lot of examples that certainly would pass through the neural network as intended, but not resemble the original training data in the slightest.

While the process of extracting information that closely resembles the original data is interesting in itself, for us it is perfectly sufficient to generate these incorrect samples. We don't need the exact image of the CEO to bypass facial recognition, we only require an image that works. Luckily, these are quite easy to generate.

We can actually train a network to do exactly this, by misusing the power of backpropagation. Recall that backpropagation begins at the back of the network and subsequently "tells" each layer how to modify itself to generate the output of the next requires. Now, if we take an existing network and simply add some layers in-front of it, we can use backpropagation to tell these layers, how to generate the inputs it needs to produce a specific output. We just need to make sure to not change the original network and only let the new layers train, as shown in Figure XXX.

For illustration, recall the assembly line example from our introduction. We have a trained assembly line that creates smart-phones from raw materials. As an attacker, we want to know exactly how much of each material this company uses to create a single smart-phone. Our approach above is similar to sneakily adding an employee to the front of the assembly and let him ask his neighbor what and how much material he should pass to him (learning through backpropagation).

Exercise 2-0: Given an image classifier, extract an image sample that will create a specific output.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/2_ExtractingInformation`

2.4 Brute-Forcing

Brute-Forcing should be reserved for when all other methods have failed. However, when it comes to Neural Networks, a lot of approaches somewhat resemble brute forcing. After all, training itself is simply showing the Network a very large set of examples. But here we are truly talking about brute-forcing a target Network over a wire, instead of training it locally.

Our scenario is as follows:

You are trying to bypass some Neural Network based security. You were able to directly attach your laptop to the camera feed, but have no access to the internal code or model.

The most obvious approach would be to find some huge database of biometric data

- Spraying?
- This is in essence adversarial

Adversarial Examples [12][20] Adversarial Competition [21] Adversarial Attacks and Defences [33]

[30] Bypassing Face Recognition using "glasses"

Blue-Team: As with password checks, try to employ the same security measures for any access control based on neural networks. Limit the amount of times a user may perform queries against the model, avoid giving hints at to what might have gone wrong, etc.

Exercise 3-0: Brute-force a neural network with a "better-than-random" approach.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/3_BruteForcing`

2.5 Neural Overflow

The next method we will cover is mathematically better than a pure brute-force approach, but not exactly the most effective. The reason we cover it is, because it is most likely one that a security expert would think of the first: "Can you overflow a neural network?".

The answer to that question is "yes". Yes, we can overflow a neural network to get a result better than pure brute-force and under special circumstances, it might even be the most feasible.

Works great for ReLU? But can also work for others

1D Example

Spiked Input

Consider the example given in Figure XXX. By randomly trying out values between 0 and 1, we have an approximate probability of 0.1 of hitting the correct value. This is quite good for brute-forcing, but we only have one input variable, so it was to be expected. If we consider n input variables and let the relative size of the peak stay the same in each dimension, we will have a probability of 0.1^n of hitting the target. For a tiny monochrome image of size 3×3, that's a probability of $0.1^9 \cong 0.0000001\%$, too low to be using pure randomness.

Now, let us take a look at the method presented here. For a single input, we have the choice of taking the value towards a high positive or a high negative number, with only one of them being correct, i.e., a probability of about 0.5 if we have no prior knowledge. We ignore the fact that the function might be perfectly level with 0, as the chance of this should be small. Thus, for n inputs, we have a probability of 0.5^n to hit the correct target. If we consider the monochrome image of size 3×3, that's a probability of $0.5^9 \cong 0.195\%$. While not great, still quite a bit better.

Blue-Team: Sanity check your inputs.

Exercise 4-0: Probe a neural network with unexpected inputs and try to gain access.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/4_NeuralOverflow`

2.6 Neural Malware Injection

- Similar to the backdoor approach

Neural Social Engineering of chatbots to translate something into ads or fake links

- Skip this exercise and "merge" it with the next?

Blue-Team: At the moment, it is near impossible to tell what a neural network does by static analysis. Before deploying any type of deep learning, at least convince yourself that it handles all the edge cases as you would expect and that it hasn't been tampered with. Any vendor, be it in security, chatbots or anything else, that is 100% certain what its neural network is doing, is lying.

Exercise 5-0: Inject malicious behavior into a chatbot.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/5_MalwareInjection`

2.6.1 AV Bypass

To bypass Anti-Virus (AV) software, one of many tactics is to employ obfuscation. Let's suppose our goal is to write some

Obviously the model file will be huge, which makes this approach somewhat unwieldy. But if our goal is to "hide" as some standard deep learning application, this should be fine.

- Anti-Malware bypass by creating a LSTM that spews out shellcode from a random number

For a more advanced implementation of these methods to convert speech into commands, we refer to the work by Lin et al. [23].

Blue-Team: Be suspicious if any software uses deep learning frameworks unexpectedly.

Exercise 6-0: Create a deep learning based plain-text-to-shellcode translator.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/6_NeuralObfuscation`

2.7 Blue Team, Red Team, AI Team

We already highlighted how deep learning can be used to do automatic penetration testing on a basic level, using reinforcement learning. That is a very complex task. We can develop much simpler red team tools using deep learning and will highlight one here: Bug hunting.

Source code has a lot of patterns. Be it architectural patterns on a large scale, such as inversion of control, or smaller patterns, such as avoiding unsafe versions of the `printf` function. Having a neural network understand these large architectural patterns is probably still too difficult, however, we can already understand simpler patterns.

To do this, we use simple techniques from natural language processing. Understanding and classifying normal english text is quite difficult and still a field of ongoing research, but source code is based on a much simpler grammar and structure. This makes it (somewhat) easier to understand for a neural network. An `if` statement is an `if` statement and doesn't have some other, metaphorical interpretation only understood from context.

As input, text classification networks mostly use a tokenized version of the text. This means, we convert words such as `printf` or `if` into numbers (`printf` = 1, `if` = 2, { = 3, } = 4, etc.) or some other representation. The neural network doesn't really care what the word `if` actually means semantically, it is only interested in where and in what patterns it appears in. Luckily, tokenizing source code is a very well understood task, as most compilers have to do it at some point [1].

The truly tricky part of designing a bug hunter is creating the training data. While it is quite straightforward to do for a single statement or code line (such as in Exercise 7-0), it gets progressively harder once you want to be able to understand bugs contextually across multiple lines and statements. This is also the domain where the neural network will surpass a simple regex-based search, as it is able to establish more context.

A good starting point for the is to synthetically generate it. This is how the, admittedly very basic, data set for the exercise was generated. Python libraries, such as NLTK, are very useful in this regard. To add some realism, this synthetic data can be augmented by interleaving safe code found in open source projects. Finally, at some point, it will make sense to add in actual vulnerable data points found in real projects. The problem here is, of course, a lot of data is needed and there aren't that many people with this type of skillset to do this rather boring task of classifying code snippets.

Blue-Team: Hoard data! Collect pcap files of compromises and even day-to-day traffic. Save potentially dangerous code that was thrown out at code reviews. All of it. Apart from training your own neural network, this data can also help a vendor craft or fit a perfect solution for your organization.

Exercise 7-0: Create a neural network for bug hunting.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/7_BugHunter`

2.8 GPU Attacks

So far, our methods were focused on attacks against the deep learning model itself and not its implementation. We can attempt to find exploits for the application found in the standard address space of the operating system. But here we find a lot of the typical mitigation techniques which makes exploitation harder.

As most deep learning frameworks use GPUs to do their calculation, it would seem obvious to check for exploitability there. If we take a look at the memory of a discrete GPU, we see that the industry has started to catch on and has implemented some of these protections there as well [26]. But we need not worry too much. While gaining code execution on the GPU might be interesting to inject cryptominers and even alter the execution of deep learning frameworks, there are easier ways for exploitation.

Textures are stored in a relatively predictable pattern, but highly dependent on the GPU manufacturer and model. Nonetheless, these textures often come in fixed sizes and the GPU allocates these chunks in those patterns automatically and continuously. Sizes of 1024×1024 for example aren't uncommon nowadays. Should a texture not need the entire space, the nearest fitting texture size is allocated. This allows the GPU to index faster, as the offsets are in deterministic increments.

But why are we talking about textures? Well, GPUs weren't exactly made for general computation (up until recently). They are still aimed at graphics processing, where the largest amount of memory is taken up by textures. When one runs general compute code, the needed buffers are stored inside of these texture arrays. As a matter of fact, that is how one did general compute in the old days, by simply uploading a "fake" texture to the GPU that actually contained values one needed to do computation.

Textures come in all shapes and sizes. We mentioned the size 1024×1024, which are 2D. But, GPUs may also store other formats, such as 1D textures. Now, 1D textures have the advantage that they can be almost any size, mostly aligned to powers of 2. These are the obvious choice for doing general compute.

So, we have a lot of texture space allocated on the GPU and the CPU has to copy parts of its memory on the RAM over to the VRAM on the GPU in order to do computations there. We also know that these allocations should all be close by in memory. Further, as texture access has to be blazingly fast, there are almost no boundary checks build into the standard languages used to program compute shaders and the code sending over the texture buffers. Developers writing this code aren't overly security conscious either, as they are focused on speed, providing us with a field day for buffer overflows. Exercise 8-0 should highlight this security flaw better than words can. Keep in mind that this exercise is written in Python, further giving a developer the illusion that he doesn't need to care about buffer overflows.

Blue-Team: Thanks to the general processing capabilities of GPUs, you basically have a second computer which you need to protect from exploitation. One, that has almost no security measures build in. Also, from a forensics view, keep in mind that crash dumps and other memory logging mostly covers the RAM on the mainboard and not VRAM. Perhaps consider dumping that too.

Exercise 8-0: Exploit a neural network running on the GPU.
→ `https://github.com/Kayzaks/HackingNeuralNetworks/tree/master/8_GPUAttack`

2.9 Supply Chain Attacks

As with any system, neural networks are susceptible to supply chain attacks. The most obvious of this would be sneaking in fake data as with did in the backdooring section. This is especially true, if one has access to some developer environment, but not production or the actual business. While code undergoes regular security checks, it is highly unfeasible that some actual person looks through the entire dataset.

But, of course, there are other methods. We've looked at some of the benchmark datasets out there. Take NSL-KDD, CICIDS2017 and related network flow data as an example. While they are difficult to create, it's not something out of scope of a security expert to accomplish. Set up an environment with simulated or even real users, run some attacks and label the `pcap` files. This is even one of the few situations where the labeling process can be automated, as everything is controlled by some fake attacker.

Now, imagine we are creating such a `pcap` dataset with labels. It's going to be bigger and better than CICIDS2017, with a bigger network, more days, up to date traffic profile. But, let's purposefully mislabel some new attack technique we developed as "normal traffic" and correctly label everything else. This process shouldn't take us too long.

Time to publish, we'll call it "CNST-2019" or similar and upload a paper describing the details on arXiv, including an example implementation of the current state-of-the-art neural network trained on our data. Perform a bit of "marketing" (blog posts and the like) and wait a while until some security vendor sends out a press release claiming "Our software was able to correctly classify 98.8% traffic found in the CNST-2019 dataset". That might be an indication that they trained on CNST-2019 and every customer of that software unable to detect our attack. If we, for some strange reason, uploaded the paper under our own name, we can always claim ignorance later on.

Obviously, this exact scenario is a complete fabrication. As noted earlier, creating synthetic `pcap` datasets isn't too difficult and a security vendor would have no reason to train on public datasets (we hope). It is more likely they use them for benchmarking. But, if it is part of their pipeline (for example, training on all public datasets in addition to their own), chances are that our secret attack will stay hidden, as it is now part of the training process to classify it as "normal traffic". The neural network was not able to generalize from other attack vectors that ours might be malicious, as it has concrete evidence it is "normal traffic". This would have not been the case, if the attack was completely absent from the dataset, as the network might be able to generalize that it is malicious.

Blue-Team: Don't simply trust public datasets for use in security critical applications. They are great for development and proof-of-concept, but be sure to double check before going into production. While it is expensive, it is probably still cheaper than creating one from scratch or risking undefined behavior.

2.10 Further Attack Vectors and Conclusion

There are countless many more attacks possible, which we haven't covered here. Hanging up adversarial images on intersections to re-route self-driving money transports. Side-channel attacks measuring the speed of a neural network to gather information about its version. Abusing federated learning to inject malicious content directly at the user level. And the list continues.

However, our goal was to give a quick overview of some of the easier to understand security risks deep learning might add to any software and simple ways of exploiting these in the context of, for example, a CTF or a penetration test. Where possible, we gave some hints for the blue team to remedy these issues.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2006.

[2] Daniel S. Berman, Anna L. Buczak, Jeffrey S. Chavis, and Cherita L. Corbett. A survey of deep learning methods for cyber security. *Information*, 10(4), 2019.

[3] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv:1712.05526*, 2017.

[4] François Chollet. Keras, 2015. URL `https://keras.io`.

[5] Chronicle. Virustotal, 2019. URL `https://www.virustotal.com/`.

[6] MITRE Corporation. Attack matrix, 2019. URL `https://attack.mitre.org/`.

[7] Balazs Csanad Csaji. Approximation with artificial neural networks. *Master Thesis*, 2001.

[8] Canadian Institute for Cybersecurity. Cybersecurity datasets. 2019.

[9] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquezb. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 28(43497):18–28, 2009.

[10] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. *AISTATS*, 2011.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[12] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[13] Aurélien Géron. A short introduction to entropy, cross-entropy and kl-divergence, 2018. URL `https://www.youtube.com/watch?v=ErfnhcEV1O8`.

[14] HDF Group. Hierarchical data format, 2019. URL `https://www.hdfgroup.org`.

[15] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405: 947–951, 2000.

[16] Horace He. The state of machine learning frameworks in 2019, 2019. URL `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`.

[17] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Rmsprop: Divide the gradient by a running average of its recent magnitude, 2012. URL `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[18] Diederik Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[19] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. *29th Australasian Joint Conference on Artificial Intelligence (AI)*, 2016.

[20] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *ICLR*, 2016.

[21] Alexey Kurakin, Ian Goodfellow, Samy Bengio, Yinpeng Dong, Fangzhou Liao, Ming Liang, Tianyu Pang, Jun Zhu, Xiaolin Hu, Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, Alan Yuille, Sangxia Huang, Yao Zhao, Yuzhe Zhao, Zhonglin Han, Junjiajia Long, Yerkebulan Berdibekov, Takuya Akiba, Seiya Tokui, and Motoki Abe. Adversarial attacks and defences competition. *NIPS*, 2017.

[22] Yann LeCun, Bernhard Boser, John S. Decker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[23] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. Program synthesis from natural language using recurrent neural networks. *UW-CSE-17-03-01*, 2017.

[24] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. *NDSS*, 2018.

[25] Michael McCloskey and Neal J.Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[26] Sparsh Mittal, S.B. Abhinaya, Manish Reddy, and Irfan Ali. A survey of techniques for improving security of gpus. *Journal of Hardware and Systems Security*, 2(3):266–285, 2018.

[27] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. *ICASSP*, 2015.

[28] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv:1802.10135*, 2018.

[29] Grant ""3Blue1Brown"" Sanderson. Deep learning, 2017. URL `https://www.youtube.com/watch?v=aircAruvnKk`.

[30] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. *Conference on Computer and Communications Security*, 2016.

[31] Nathan Shone, Tran Nguyen Ngoc, Vu Dinh Phai, and Qi Shi. A deep learning approach to network intrusion detection. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(1), 2018.

[32] Kouichi Yamaguchi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. A neural network for speaker-independent isolated word recognition. *ICSLP*, pages 1077–1080, 1990.

[33] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. Adversarial examples: Attacks and defenses for deep learning. *arXiv:1712.07107*, 2017.