BLACK CELL

Protecting critical infrastructures

Domain Name Generating Algorithms Detection

Table of Contents

1. Domain name generating algorithms 3

2. Detection methods 4

2.1. Entropy 4

2.2. Feedforward neural networks 6

2.2.1. Fully connected neural networks 9

2.2.2. Convolutional neural network 10

2.3. Recurrent neural networks 11

2.3.1. Fully recurrent neural networks 12

2.3.2. Long short-term memory 13

2.3.2. Bidirectional recurrent neural networks 14

2.4. Hybrid architecture 15

3. Summary 16

1. Domain Name Generating Algorithms

Domain Generating Algorithms (DGA) are a tool widely used by malware developers, making it harder to detect or interfere with their malware's communication. Malware infections would be quite short-lived if they were to use hardcoded domains or IP addresses for phoning home and receiving commands. Security personnel could simply dump the hard coded domains and pre-emptively feed them into a network blacklisting appliance in an attempt to restrict outbound communication from infected hosts within a victim's infrastructure. To avoid this, DGAs are used to generate hundreds or even tens of thousands (of which only a subset is actually used for communication) of unique domains a day that can be used as rendezvous points with the C&C server.

DGA's are symmetric in the sense, that domains generated by the malware will be (in most cases) identical to the domains generated by the C&C server. The algorithms use some sort of seed to generate domains. For example, earlier versions of the CryptoLocker malware family used the current date to generate domains. Below you can see a Python implementation of the DGA used by these early versions of CryptoLocker. The algorithm takes a date, performs a number of bitwise operations on the integers that make up a date, finally converting the produced integers into the characters that make up the final domain name.

```
def generate_domain(year, month, day, length=32, tld=''):
  domain = ""
  for i in range(length):
    year = ((year ^ 8 * year) >> 11) ^ ((year & 0xFFFFFFF0) << 17)
    month = ((month ^ 4 * month) >> 25) ^ 16 * (month & 0xFFFFFFF8)
    day = ((day ^ (day << 13)) >> 19) ^ ((day & 0xFFFFFFFE) << 12)
    domain += chr(((year ^ month ^ day) % 25) + 97)

  domain += tld
  return domain


>> generate_domain(2021, 10, 12, tld='.com')
>> ovyvwnkjserklcrjwwhcpucyurwjaelg.com
```

CryptoLocker DGA with sample output.[1]

Malware utilizing DGAs are able to generate a large number of domains daily, therefore it is infeasible to create a CTI (Cyber Threat Intel) feed containing the newly generated domains of each new malware family. Instead, we need to consider more sophisticated techniques that are able classify whether or not a domain was produced by a DGA.

[1] *Taken from: https://github.com/endgameinc/dga_predict/blob/master/dga_classifier/dga_generators/cryptolocker.py*

2. Detection Methods

DGA detection methods have been somewhat of a hot topic in recent months and years. As mentioned above, we cannot simply create a set of all possible generated domains and blacklist them. Instead, we need to consider methods that can look at any given domain and determine whether or not it was generated by a DGA. There has been much discussion among Cybersecurity and Data Science experts about the best method of classifying these domains. In the following sections of this paper, we will be looking at a relatively simple approach leading into more complex neural network-based approaches, including our own chosen method.

2.1. Entropy

Let's take another look at the example domain produced by the previously discussed DGA.

ovyvwnkjserklcrjwwhcpucyurwjaelg.com

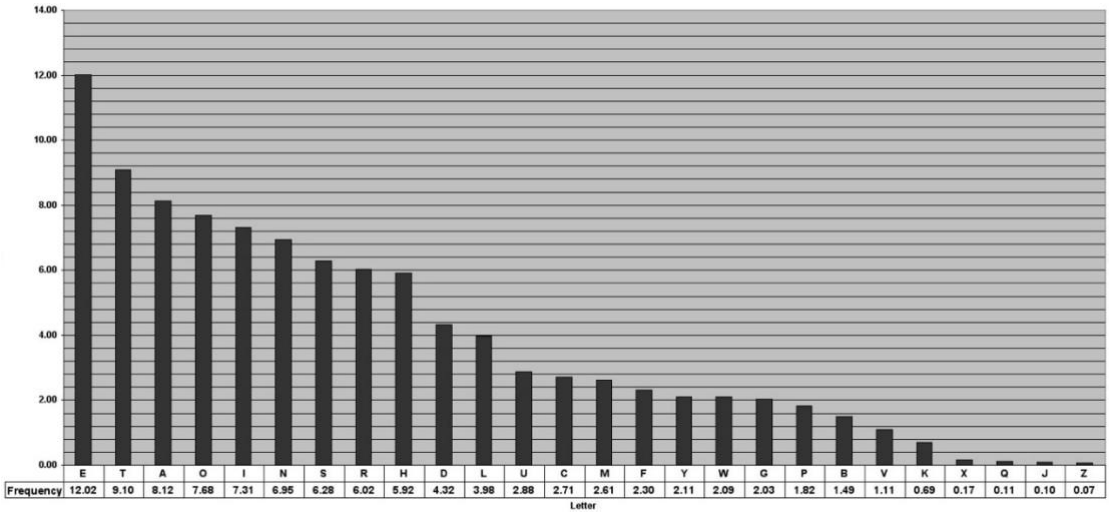
A CryptoLocker domain.

Just by looking at this domain we can instinctively tell something isn't right. It's almost as if someone "mashed" their keyboard when registering a domain name. We get this feeling because each of the characters that make up the domain seem randomly chosen.
In contrast, when we see a domain that consists of words from a natural language, then we don't get the sense that it was randomly generated, even if the words are from a language we are unfamiliar with.

lecanardenchaine.fr

The domain of a French satirical newspaper.

The letters that form words in natural languages are not randomly selected. In fact, it is really easy to calculate the probability of a given letter occurring in a sequence, meaning that words in a natural language aren't very random.



A letter frequency chart for the English language.[2]

[2] *Taken from: http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html*

As such, if we could simply quantify the "randomness" of a domain, then we could figure out a "boundary" above which we can assume a domain was randomly generated. Thankfully we are able to quantify the "randomness" of a string, with the concept of Shannon's entropy.

Shannon's entropy quantifies the amount of "information" or "surprise" in a variable. If a domain contains the letter "E" a lot, its entropy will likely be lower, because "E" is the most common letter in the English language, therefore it should occur more often and its presence will be less of a surprise. On the other hand, if the domain contains a "J"s or a "Z"s, its entropy will likely be higher as these letters are uncommon.

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log \mathrm{P}(x_i)$$

Shannon's entropy formula.

This is all well and good, but how does it perform in the real world? Using a real-world dataset containing both regular and algorithmically generated domains, we calculated the entropy of each domain, and found an optimal boundary value for classifying the domains. However, with this method we were only able to achieve an achieve an accuracy between 60-65%. That's only 10-15% better than a random guess. But why does this method of classification perform so poorly?

The truth is if it were this simple to detect DGAs, no one would use them. As with any other technology, the creators of these algorithms created newer and more advanced methods of generating domains. The previous CryptoLocker example is only one of many families of DGAs. Other DGAs found a relatively simple way around the method we described above, with the use of dictionaries. The below domains are examples generated by the Suppobox family of DGAs. As you can see, they could easily be legitimate domains.

journeyready.net

wouldinstead.net

sickhurry.net

darkhope.net

cloudthirteen.net

dutybegan.net

christianaashleigh.net

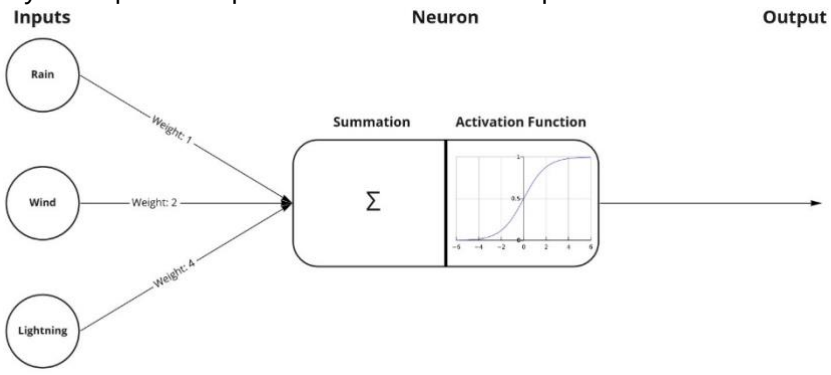
Example Suppobox domains.

These domains are generated by selecting random words from a dictionary file and combining them together. Because they are made up of regular natural language words, they do not exhibit the randomness of the previous examples. Therefore, our previous entropy-based detection method is not well suited to detect this different class of DGAs.

To detect these more sophisticated DGAs, we need to find some more advanced detection methods that look at more features of the domains than just entropy. There are many approaches, but for our own detection capabilities we chose neural networks.

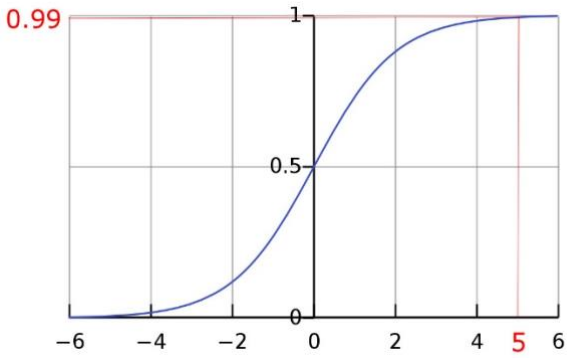
2.2. Feedforward Neural Networks

Many of us will have heard a lot about neural networks and all of their fantastic applications. Let's see if we could use neural networks to build better detection methods for these more sophisticated DGAs. But what exactly are neural networks? They are made up of a number of interconnected artificial neurons, modelled on biological neural networks such as those found in animal brains. This sounds quite esoteric, but at its core it's really quite simple. These artificial neurons simply take the sum of some given inputs and pass it through a mathematical function (sometimes a simple curve), producing an output. The output of the neuron is then weighted and connected as the input to other neurons. Image the weight applied to the output as a sort of "importance" or how much this output should contribute to the final result produced by the network. Hopefully a simple example will clear all of this up.



A simple artificial neuron.

If you look at the above illustration you'll see a number of inputs, corresponding weightings, an activation function and an output. Let say we want our neuron to be able to decide if the current weather constitutes a storm or not. It will take values for the amount of rain, wind and lightning currently observable. Rain doesn't necessarily imply a storm therefore we assign it a low weight. Wind however is a better indicator of a potential storm; therefore, we assign it a higher wight. Finally, if there's lightning, we can be fairly certain there is a storm, therefore we assign it the highest weight. For the sake of this example, we will have the input values on a scale -1 to 1. For example, our rain sensor produces a value of -1 when it is completely dry and 1 when it is completely wet. Let say we currently have 0.5 units of rain, 0.75 units of wind and 0.75 units of lightning. After applying the weights, we have 0.5 units of rain, 1.5 units of wind and 3 units of lightning, totalling 5 units. We pass this total into our activation function (in this case a sigmoid function), to get our result.

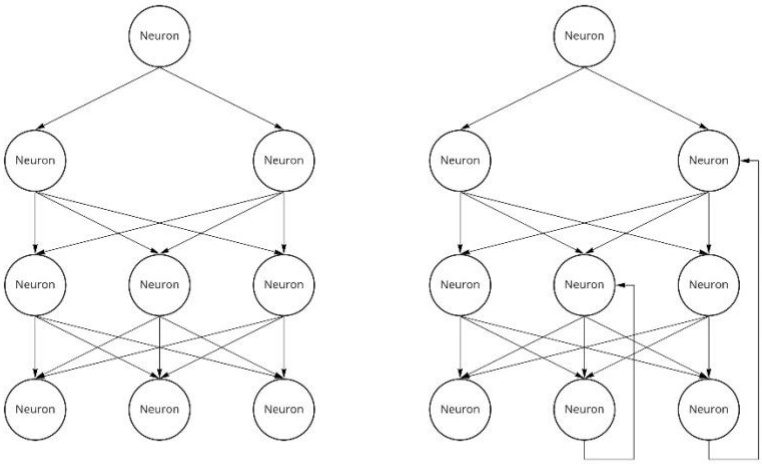


Sigmoid activation function with results.

As you can see the activation produces a result of 0.99 (on a scale of 0 to 1) indicating we are very certain there is a storm. With different inputs such as -0.75 units of rain, 1 units of clouds and -1 units of lightning we get an output of 0.06 indicating there is no storm.

In our example we get reasonable outputs for each input. However, if it were to produce inaccurate results, we could simply optimize the weightings on the inputs. This is where we begin to introduce the concept of machine learning. If we have a collection of rain, wind and lightning values with corresponding results we would like to see, we can begin finding the optimum weights that will produce the desired output for each input. This is essentially how we train neural networks. With a single neuron this process is somewhat meaningless, but once we begin connecting a network of these neurons, training the network will be very important.

Now that we have an understanding of how neurons work, lets see how we can arrange these neurons in a network and begin to make predictions on whether or not a domain was generated algorithmically. Soon, we will look at some feedforward neural networks and the results we can achieve with them. The term "feedforward" simply means the connections between the neurons do not form a cycle.



An acyclic arrangement (left) and a cyclic arrangement (right) of neurons.

In our previous storm assessment example, we used a neuron to create a very simple relationship between three inputs and an output. If we begin adding more of these neurons with even more inputs, we can start building increasingly more complex relationships between

the various features of our input data. These relationships can become highly intricate, non-linear and difficult to replicate by other means.

The optimizing or "training" of large networks happen similarly to our aforementioned neuron example. We give the network a number of inputs and an expected output and initialize the weights between the neurons randomly. The network takes an input, passes it through each neuron and calculates the output. We call this process forward propagation. We then compare the produced result with our expected result and calculate the error produced by our network. We can then calculate how much each individual neuron contributes towards the total error produced and optimize the weights between the neurons accordingly. In general, networks are optimized by performing gradient descent on the loss function (a function of the error produced by the network) in order to find the minimum possible loss, indicating we have found the optimal solution for predicting outputs. This process is called backwards propagation and it's done over many iterations with many input-output pairs.

It is important to note that through this process we aren't finding the ultimate all-knowing solution to predicting something, but rather we are finding the best solution to map our previously collected inputs to our outputs. This doesn't necessarily mean if we give the network previously unseen inputs, it will produce the correct output every time. This is where things start to become tricky, because the lowest possible loss does not mean the network is generalizable for other inputs. If our network perfectly predicts the solutions for each input during training, but often gets previously unseen inputs wrong, we are experiencing a problem called over-fitting.

Because neural networks learn from the data we feed it, we need to be sure we create a good dataset. Remember, "garbage in, garbage out". In most cases want our dataset to be as large as possible and for the data to be a good representation of all possible inputs.

We now have a pretty good high-level understanding of how neural networks work. But we still need to discuss a few things before we can begin predicting DGAs. If we think back to our storm example we used a bunch of numeric inputs, but with DGAs we only have text, which we cannot simply pass into a neural network. First, we need to do some pre-processing to turn our text into numeric values. We could do something simple like use the ASCII values of each character, but this probably won't do us much good.

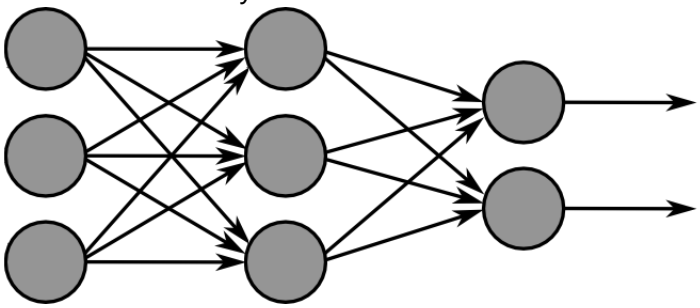
If we think back to the storm example, we used attributes of the weather such as the amount of rain, instead of passing in the exact meteorological state of the world. Similarly, we want to not only turn domains into numerical values, but we also want to it in a way that preserves information we want to learn from, or even extract certain features of the domains whilst getting rid of unimportant information. One of the features that could be interesting to use in a neural network is the topic we started this paper with, entropy.

If we were to go into all the computation linguistic methods of extracting features from text, we would be straying dangerously far from the topic at hand. For our examples we will be simply converting domains into n-grams of letters and one-hot encoding them. N-grams are a computational linguistic model, dealing with sequences of n items, where the items could be things like words or symbols. For the purposes of this paper, it is enough if we know our

domains are broken down into numerical values, that preserve the information contained in the original string.

2.2.1. Fully Connected Neural Networks

Now that we know how to process our domains, the last consideration we should discuss is how we arrange and connect the neurons in our network. Neurons in a network are typically arranged in layers with the neurons in each layer connected to each subsequent layer in some fashion. The easiest way we can connect neurons would be to simply connect every neuron in one layer to each neuron in the next layer.



A fully connected neural network.[3]

This is what we call fully connected neural networks. We now know everything we need to construct our neural network in our framework of choice. In our testing we were able to achieve between 79-81% accuracy with a simple fully connected neural network with only 2 hidden layers. Below you can find the TensorFlow output for the first 5 epochs of training. Epochs are the number of times the whole dataset has been processed during training.

```
Epoch 1/15
38462/38462 [==============================] - 223s 6ms/step - loss: 0.5188 - accuracy: 0.7323
Epoch 2/15
38462/38462 [==============================] - 227s 6ms/step - loss: 0.4767 - accuracy: 0.7662
Epoch 3/15
38462/38462 [==============================] - 228s 6ms/step - loss: 0.4549 - accuracy: 0.7825
Epoch 4/15
38462/38462 [==============================] - 232s 6ms/step - loss: 0.4400 - accuracy: 0.7932
Epoch 5/15
38462/38462 [==============================] - 228s 6ms/step - loss: 0.4285 - accuracy: 0.8007
```

Training data for a fully connected neural network.

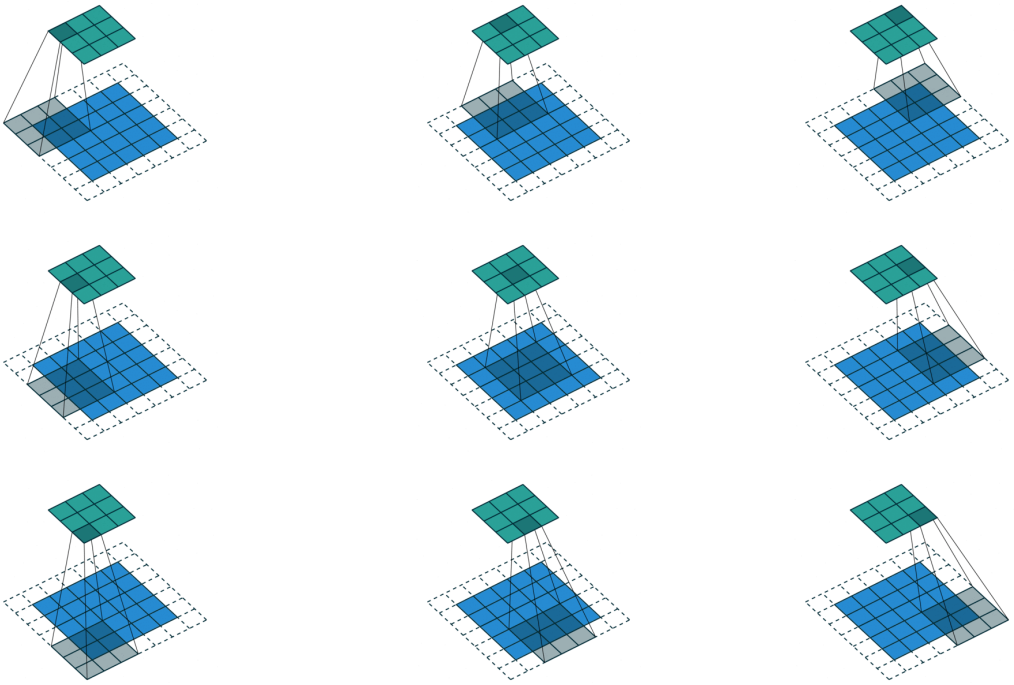
This is a huge improvement over a purely entropy-based approach, but there is still a lot of room for improvement. Fully connected networks can yield good results depending on the

[3] *Adapted from: https://upload.wikimedia.org/wikipedia/commons/e/e1/MultiLayerNeuralNetwork.png*

application, but are prone to overfitting and there are plenty of more sophisticated models we can explore.

2.2.2 Convolutional Neural Network

With our fully connected neural network we were able to achieve decent results, but let's see if we can find some better ways in which we can arrange and interconnect the neurons in our network. One of the methods we can employ is the use of kernel filters. Instead of connecting every neuron together, we can shift a filter of a certain size (3x3 for example) across the input layer and map the numerous inputs covered by the filter to a single (or multiple) neuron(s) in the next layer.



An example of a kernel filter moving across the input data in a convolutional neural network.[4]

As you may be able to guess from the above figure, these networks are especially well suited to image recognition tasks. This is because the use of kernel filters puts a greater emphasis on very localized features of an image instead of focusing on an image as a whole. Often, we also include a number of fully connected layers at the end of convolutional models, to both make sure we can map to the desired output size and also to allow for interconnection of the results of the convolutional layers.

Convolutional networks are also sometimes used in natural language processing, because much like with image processing, our network can put emphasis on subsections of the input

[4] *Taken from: https://commons.wikimedia.org/wiki/File:Convolution_arithmetic_-_Padding_strides.gif*

text. We also do not necessarily need to use 2 dimensional inputs as convolution can also be applied in 1D.

In our testing we created a simple convolutional network with a fully connected layer at the end and used the same dataset for training as we did with the fully connected network. Although we didn't spend much time optimizing our model, it was clear from the start, that we weren't going to get very good results. In our testing we were only able to achieve an accuracy of up to 65%. Not much better that entropy alone. It seems our domains are better analysed as a whole, probably because they are not big enough, therefore the spatial structure of our data is less important.

```
Epoch 1/15
38462/38462 [==============================] - 515s 13ms/step - loss: 0.6435 - accuracy: 0.6449
Epoch 2/15
38462/38462 [==============================] - 511s 13ms/step - loss: 0.6413 - accuracy: 0.6457
Epoch 3/15
38462/38462 [==============================] - 513s 13ms/step - loss: 0.6400 - accuracy: 0.6458
Epoch 4/15
38462/38462 [==============================] - 513s 13ms/step - loss: 0.6389 - accuracy: 0.6459
Epoch 5/15
38462/38462 [==============================] - 515s 13ms/step - loss: 0.6381 - accuracy: 0.6460
```

Training data for a convolutional neural network.

2.3. Recurrent Neural Networks

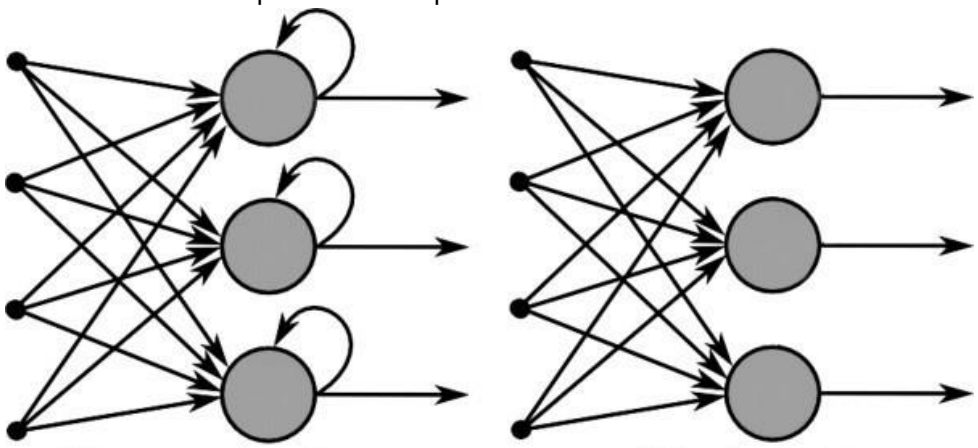
You might be wondering why we made the distinction between cyclic and acyclic networks earlier. There is a very good reason for this, but first we need to understand the processing of temporal data. Staying on the theme of weather prediction, lets imagine we want to predict whether or not it is going to rain soon, based on the position of storm clouds. If we have data that shows us that a cloud's position is moving towards us, the probability of rain increases. But if we take a snapshot of this continuous data, the cloud is stationary and we would need to essentially guess which way it is moving. We need to somehow use the information from previous data samples to be able to infer where a cloud is moving. To do this with neural networks we essentially need to add memory to neurons, to be able to remember information from previous time steps. The data added in to subsequent time steps is called the hidden state.

Text can also be sequential data. Think of the sentence "Is it raining?". Looking at any of the words individually, we cannot determine the intention of the text. With the word "is" we know we are talking about the existence of something. When we retain memory of the word "is" and combine it with the word "it" we can deduce we are questioning the existence of something. If we remember the last two words and combine it with the word "raining" we now know the meaning of our sentence. If we were looking at the word "raining" without the context of the previous two words, we wouldn't know we were asking about the presence of rain.

This ability to learn contextual information based on previous characters will be incredibly useful when we want to detect DGAs generated with the use of dictionaries.

2.3.1 Fully Recurrent Neural Networks

We can implement this memory in neural networks in a number of ways, but one of the simplest examples are fully recurrent neural networks. Here each neuron simply passes information to every other neuron in subsequent time steps.



A visualization of a recurrent neural network.[5]

Unfortunately, as standard RNN's are particularly popular, TensorFlow doesn't have optimized GPU kernels for this specific type of neural network, therefore we had to restrict the training of this neural network somewhat. Nonetheless, with a simple fully recurrent neural network we were able to achieve accuracies of up to 91%.

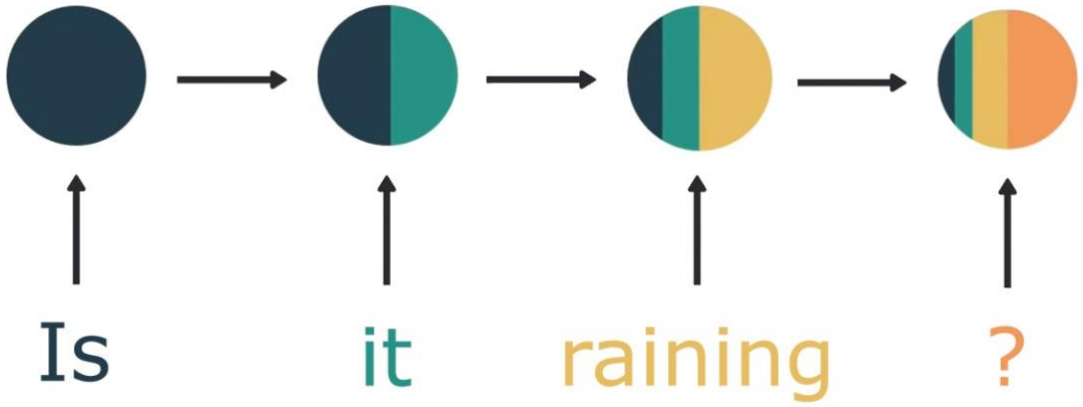
```
Epoch 1/15
4425/4425 [==============================] - 886s 195ms/step - loss: 0.3288 - accuracy: 0.8507
Epoch 2/15
4425/4425 [==============================] - 865s 196ms/step - loss: 0.2734 - accuracy: 0.8829
Epoch 3/15
4425/4425 [==============================] - 864s 195ms/step - loss: 0.2486 - accuracy: 0.8951
Epoch 4/15
4425/4425 [==============================] - 864s 195ms/step - loss: 0.2297 - accuracy: 0.9037
Epoch 5/15
4425/4425 [==============================] - 864s 195ms/step - loss: 0.2176 - accuracy: 0.9092
```

Training data for a fully recurrent neural network.

[5] *Adapted from: https://commons.wikimedia.org/wiki/File:RecurrentLayerNeuralNetwork.png*

2.3.1 Long Short-Term Memory

The results we were able to achieve with fully recurrent networks were good, but there is a problem going on behind the scenes. The information passed from one timestep to the next, is the aggregation of all the information from previous timesteps. As you can see on the below image, the information from the first iteration is less and less predominant with each iteration. For example, with a really long question, by the end of the sentence we can barely remember the sentence started with a "what", making it difficult to deduce the intention of a sentence.



A visualization of short-term memory.[6]

This problem is called short-term memory, which is caused by the vanishing gradient problem during backpropagation. The impact of the information learned in earlier iterations, decreases exponentially with each subsequent iteration. To combat this we can add gates, to the neural network which store the hidden state over multiple timesteps. This way, information learned in the first timestep can be reintroduced in much later timesteps, preventing this information from being diminished or even lost.

Using a long short-term memory neural network, we were able to achieve accuracies up to 93%. These results are fantastic, but we can still make improvements.

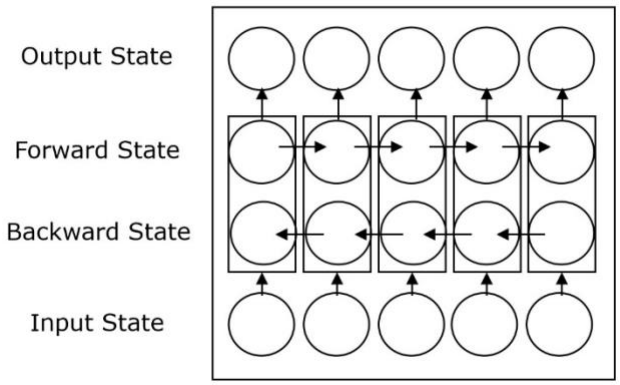
```
Epoch 1/15
38462/38462 [==============================] - 646s 17ms/step - loss: 0.3196 - accuracy: 0.8512
Epoch 2/15
38462/38462 [==============================] - 638s 17ms/step - loss: 0.2366 - accuracy: 0.8968
Epoch 3/15
38462/38462 [==============================] - 637s 17ms/step - loss: 0.2055 - accuracy: 0.9110
Epoch 4/15
38462/38462 [==============================] - 638s 17ms/step - loss: 0.1867 - accuracy: 0.9193
Epoch 5/15
38462/38462 [==============================] - 637s 17ms/step - loss: 0.1715 - accuracy: 0.9262
```

Training data for an LSTM neural network.

[6] *Adapted from: https://www.youtube.com/watch?v=LHXXI4-IEns*

2.3.2 Bidirectional Recurrent Neural Networks

As we discussed earlier recurrent neural networks are great because they allow for information to be remembered. But with the examples we looked at so far neurons can only learn from previous timesteps, but not future ones. But what if they could? To do this we can use something called a bidirectional recurrent neural network. These networks use two layers that process the input in different directions in time to produce one single output.



A visualization of a bidirectional recurrent neural network.[7]

As you can see in the above image, one layer processes the text in a positive time direction, and another layer processes the text in a negative time direction. We then combine the results from the two layers to produce an output. With this architecture, neurons are able to use contextual information from both directions. We can also use the previously discussed long short-term memory in this architecture as well to help with the vanishing gradient problem.

With a bidirectional long short-term memory neural network, we were able to achieve 94% accuracy. These results are nearing a production ready model, that we could use to reliably detect DGAs.

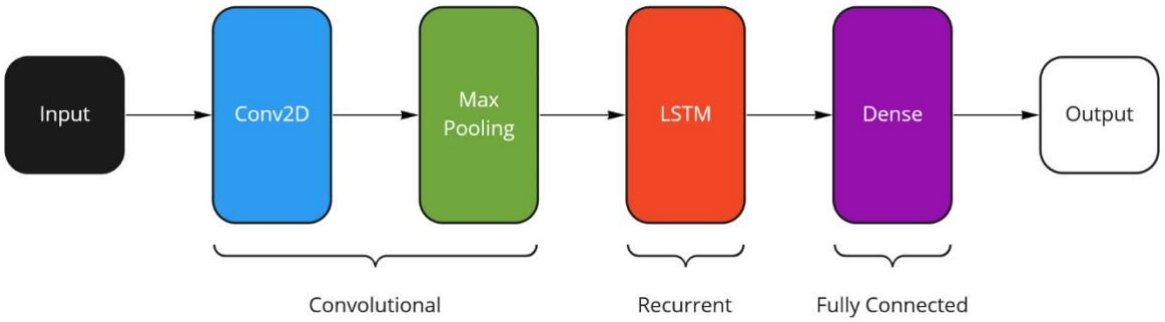
```
Epoch 1/15
38462/38462 [==============================] - 927s 24ms/step - loss: 0.3185 - accuracy: 0.8621
Epoch 2/15
38462/38462 [==============================] - 913s 24ms/step - loss: 0.2326 - accuracy: 0.9083
Epoch 3/15
38462/38462 [==============================] - 920s 24ms/step - loss: 0.2006 - accuracy: 0.9223
Epoch 4/15
38462/38462 [==============================] - 920s 24ms/step - loss: 0.1804 - accuracy: 0.9301
Epoch 5/15
38462/38462 [==============================] - 920s 24ms/step - loss: 0.1656 - accuracy: 0.9377
```

Training data for a bidirectional LSTM neural network.

[7] *Adapted from: https://commons.wikimedia.org/wiki/File:Structural_diagrams_of_unidirectional_and_bidirectional_recurrent_neural_networks.png*

2.4. Hybrid architecture

We have discussed a lot of neural network architectures with varying results, but please remember that these architectures apply to layers that we can mix and match as we please. For example, there's nothing stopping us from putting convolutional layers in front of a long short-term memory layer.



A neural network with convolutional, long short-term memory and fully connected layers.

You can experiment by combining any number of different layers to achieve the best results possible. This is exactly what we did to create our own in-house detection capabilities. Surprisingly, we ended up using convolutional layers with a number of other layers, even though in our earlier test convolutional networks performed the worst. The combination of layers that perform the best for your dataset may very well surprise you. With our combination of various neural network architectures, we were able to achieve an accuracy of up to 98%, however we found that models with accuracies between 96-97% were more generalizable and did not suffer too much from overfitting. Below you can see our results with a larger training dataset over the first 5 epochs.

```
Epoch 1/15
196713/196713 [==============================] - 4719s 24ms/step - loss: 0.1662 - accuracy: 0.9359
Epoch 2/15
196713/196713 [==============================] - 4718s 24ms/step - loss: 0.1298 - accuracy: 0.9514
Epoch 3/15
196713/196713 [==============================] - 4720s 24ms/step - loss: 0.1182 - accuracy: 0.9563
Epoch 4/15
196713/196713 [==============================] - 4721s 24ms/step - loss: 0.1118 - accuracy: 0.9589
Epoch 5/15
196713/196713 [==============================] - 4728s 24ms/step - loss: 0.1076 - accuracy: 0.9605
```

Training data for our custom neural network.

3. Summary

In this paper we had a look at what domain generating algorithms are and why they are used. We looked at the domains' structure and some basic approaches for detecting them. We then dove into the world of neural networks to see how they work and how good they are at detecting DGAs. We went into quite a bit of depth, but naturally we had to omit some of the gritty details. Hopefully you now have a better understanding of DGAs and neural networks. Below you can find a summary of the results we were able to achieve. Of course, there are many (often better) metrics than simple accuracy, but for now it should give us a good indication of a model's performance.

Entropy	FCNN	CNN	FRNN	LSTM	BiLSTM	Combination
60-65%	79-81%	64-65%	90-91%	92-93%	93-94%	96-98%

Accuracy results for the various neural network architectures we had a look at.

When developing our own model, we worked with many publicly available, closed source and custom datasets during training. Our datasets in total took tens of gigabytes of disk space and contained more than a 150 million domain names. One of the datasets we used for validation contained samples from 92 known malware families. Below you can see a table containing the validation results for the aforementioned malware families.

DGA Family	Accuracy	DGA Family	Accuracy	DGA Family	Accuracy	DGA Family	Accuracy
bamital	100.00%	pandabanker	99.99%	feodo	100.00%	suppobox	99.74%
banjori	99.97%	pitou	65.49%	fobber	98.70%	sutra	99.31%
bedep	99.40%	proslikefan	93.81%	gameover	99.92%	symmi	87.69%
beebone	100.00%	pushdotid	95.98%	gameover_p2p	99.99%	szribi	94.54%
blackhole	100.00%	pushdo	90.12%	gozi	95.86%	tempedrevetd d	96.23%
bobax	98.00%	pykspa2s	99.06%	goznym	91.76%	tempedreve	96.08%
ccleaner	100.00%	pykspa2	99.34%	gspy	100.00%	tinba	99.44%
chinad	99.79%	pykspa	97.47%	hesperbot	94.38%	tinynuke	99.63%
chir	100.00%	qadars	99.68%	infy	99.84%	tofsee	98.40%
conficker	97.10%	qakbot	99.45%	locky	94.11%	torpig	89.89%
corebot	99.64%	qhost	60.87%	madmax	99.74%	tsifiri	100.00%
cryptolocker	99.43%	qsnatch	42.93%	makloader	100.00%	ud2	100.00%
darkshell	87.76%	ramdo	99.98%	matsnu	74.42%	ud3	95.00%
diamondfox	76.96%	ramnit	97.67%	mirai	95.71%	ud4	91.00%
dircrypt	97.83%	ranbyus	99.75%	modpack	86.88%	urlzone	98.67%
dmsniff	91.00%	randomloader	100.00%	monerominer	99.99%	vawtrak	94.85%
dnsbenchmark	100.00%	redyms	100.00%	murofetweekly	99.99%	vidrotid	98.33%
dnschanger	97.20%	rovnix	99.83%	murofet	99.79%	vidro	97.40%
dyre	99.92%	shifu	97.90%	mydoom	93.65%	virut	97.69%

ebury	99.95%	simda	97.49%	necurs	97.39%	volatilecedar	94.18%
ekforward	99.73%	sisron	100.00%	nymaim2	67.74%	wd	100.00%
emotet	99.88%	sphinx	99.73%	nymaim	91.32%	xshellghost	100.00%
omexo	100.00%	padcrypt	99.33%	oderoor	97.92%	xxhex	100.00%

The prediction accuracies of our model, per malware family.

As a comparison we decided to test the accuracy of the simple entropy-based detection method we started this paper with. We took the same dataset, we trained our neural network with and created a balanced version of it, meaning there were equal DGA and non-DGA domains. This is important because an unbalanced dataset would very heavily skew the results in the case of the entropy-based method. With neural networks it's less important to use a balanced dataset, as it can be both a benefit and a detriment depending on the scenario. In most cases you should start with an equal number of samples (where possible), but you can change the balance of the different classes as you see fit. We then calculated our optimal boundary value and used it to make predictions for our validation dataset.

DGA Family	Accuracy	DGA Family	Accuracy	DGA Family	Accuracy	DGA Family	Accuracy
bamital	97.40%	pandabanker	38.69%	feodo	89.58%	suppobox	13.37%
banjori	77.43%	pitou	0.01%	fobber	56.20%	sutra	57.33%
bedep	78.26%	proslikefan	5.96%	gameover	99.99%	symmi	43.93%
beebone	40.95%	pushdotid	15.42%	gameover_p2p	99.55%	szribi	4.90%
blackhole	80.02%	pushdo	10.13%	gozi	58.97%	tempedrevetdd	9.20%
bobax	47.67%	pykspa2s	25.92%	goznym	15.11%	tempedreve	20.10%
ccleaner	19.23%	pykspa2	26.37%	gspy	63.27%	tinba	33.36%
chinad	96.40%	pykspa	18.66%	hesperbot	43.82%	tinynuke	99.23%
chir	51.00%	qadars	78.16%	infy	10.35%	tofsee	0.00%
conficker	9.83%	qakbot	72.14%	locky	39.63%	torpig	6.94%
corebot	95.16%	qhost	26.09%	madmax	37.89%	tsifiri	0.00%
cryptolocker	63.56%	qsnatch	0.14%	makloader	100.00%	ud2	93.93%
darkshell	0.00%	ramdo	16.18%	matsnu	46.87%	ud3	88.33%
diamondfox	2.92%	ramnit	54.79%	mirai	67.14%	ud4	4.00%
dircrypt	56.96%	ranbyus	70.03%	modpack	9.38%	urlzone	64.79%
dmsniff	4.00%	randomloader	20.00%	moneerominer	78.20%	vawtrak	10.19%
dnsbenchmark	100.00%	redyms	67.65%	murofetweekly	100.00%	vidrotid	32.67%
dnschanger	18.65%	rovnix	97.87%	murofet	68.08%	vidro	37.65%
dyre	98.60%	shifu	7.16%	mydoom	0.64%	virut	0.00%
ebury	85.20%	simda	3.12%	necurs	53.97%	volatilecedar	65.86%
ekforward	0.00%	sisron	11.82%	nymaim2	36.40%	wd	99.79%
emotet	77.75%	sphinx	80.71%	nymaim	12.63%	xshellghost	54.00%
omexo	100.00%	padcrypt	3.77%	oderoor	13.92%	xxhex	0.00%

The prediction accuracies of a simple entropy-based detection method, per malware family.

As you can see the results are substantially worse than our neural network model, and as we discussed earlier in the paper, it cannot handle natural language domains. It also struggles with

very short domains, where there is not enough information to make a good prediction and the accuracy begins to devolve to a random guess or even worse.

We often associate machine learning with graphics cards or even tensor processing units, and you may assume that our detection method would consume a load of resources to make predictions. However, this is not really the case. We tested the throughput of our Splunk search command implementation and summarized the results you can expect with regular server hardware below. As you can see no special hardware is required to run these detection methods. Keep in mind that these throughputs are measured with unique domains. In a real-world scenario, with deduplication and a whitelist, you will struggle to saturate even a single vCPU.

vCPU	Minimum RAM	Throughput
1	2 GB	~ 28.000 domains / minute
2	3 GB	~ 62.000 domains / minute
4	4 GB	~ 108.000 domains / minute
8	4 GB	~ 188.000 domains / minute
16	7 GB	~ 300.000 domains / minute

We use these very methodologies to detect DGAs in our Fusion Center. We research and develop many other tools, both in the field of machine learning and other new technologies, to help secure our client's infrastructures. To find out more about our Fusion Center and how it differs from a regular SOC, visit our Fusion Center website.