Artificial Intelligence for Cybersecurity

Mohan Santokhi, Jay Santokhi
{mohan, jay}.santokhi.gmail.com

Abstract

There is a tremendous interest in the cybersecurity arena with regards to use of Artificial Intelligence (AI), particularly in areas such as Network Intrusion Detection (NID), Malware Detection, Phishing Detection and Security Surveillance, however replicating the success of image classification, speech analysis, speech synthesis and machine translation is proving to be more challenging for Cybersecurity. The challenge is associated with the lack of public datasets and appropriate feature engineering rather than AI techniques. One may intuitively deduce that datasets for Cybersecurity yield more diversity in the data features. Given this increased variation in dataset features, common views on what AI can realistically achieve for Cybersecurity use cases is often misconstrued. As a result, we need to be more cautious when assessing vendors' claims and not to be 'blown away' by the hype. This paper provides an introduction to AI concepts and models which will be helpful in assessing said vendors' products. Security Surveillance, especially, has benefited greatly from the success of image classification and object detection. Open source trained models are available to construct embedded AI based Security Surveillance systems.

1. Introduction

With ever increasing success of Artificial Intelligence (AI) in the field of computer vision, speech to text, text to speech, machine translation and language modelling, the question is: *Can these AI techniques be applied to benefit Cybersecurity and what are the challenges that lay ahead*?

One thing that distinguishes Cybersecurity from successful domains mentioned above is the lack of modern datasets available for Cybersecurity AI researchers and practitioners. As we will see, datasets are the fuel that drives successful application of AI.

The first component of building a neural network is to gather our initial dataset. This is the most crucial and time-consuming task, particularly for Cybersecurity use cases, as we will see when discussing Network Intrusion Detection (NID) systems and Malware Detection. For computer vision and other domains, it is a lot easier to 'have a feel' for the data features. For example, in image classification we have images consisting of width, height and colour channels (RGB pixels) as well as the category or class of images. For speech to text we have a 1-Dimensional speech signal and associated labels. For machine translation and language modelling we have a set of documents.

For Cybersecurity use cases there is much more diversity than people intuitively expect, which leads to misconceptions about what anomaly detection technology can realistically achieve. Even with a single network, the most basic characteristics such as bandwidth, duration of connections, and application mix can exhibit immense variability. For malware classifiers one needs a dataset which has hundreds of thousands of malware files, this can however be achieved but which features of the malware do we select and ignore? Thus, deciding on which feature to capture is a difficult and time-consuming task for cybersecurity.

Currently in the Cybersecurity arena there are increasing number of vendors who claim their product use AI however details on datasets used and models created is unknown, which is understandable considering they have invested a great deal of time and effort. In Cybersecurity it can said that datasets are intellectual property compared to other domains, however we need to be cautious with these vendors' claims and not be 'blown away' by the hype, but at the same time we need to have an open mind and make objective judgements based on some knowledge on how AI models are developed and tested.

We should also be cautious of claims made in the research papers which are usually based on small and old datasets which are not particularly relevant today, notably the NSL-KDD dataset. The NSL-KDD dataset has been used in large number of research papers to demonstrate AI models with high accuracy for NIDs. The NSL-KDD dataset is based on the KDD-1999 dataset with few recent improvements and should be treated as a '*hello world*' dataset to illustrate dataset feature engineering, as we will do in section 5.

The goal of this paper is to present high-level AI concepts such as Artificial Neural Networks (ANNs), different methods of training ANNs, dataset feature engineering as well as presenting different ANN architectures through

Cybersecurity use cases. Use cases we will be discussing include Network Intrusion Detection (NID), Malware Detection, Phishing Detection and Security Surveillance. In each of the use cases we well also highlight the current challenges faced in the industry today.

2. AI Overview

"*The goal of AI is to develop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future outcomes of interest*."

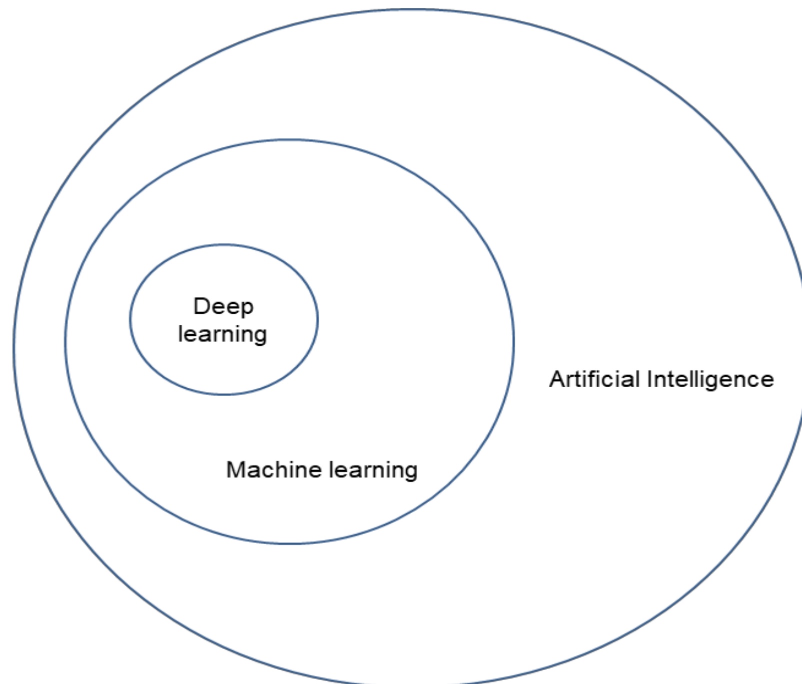

Figure 1: A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence

AI was born in the 1950s through Rosenblatt's invention of the perceptron [1] and Symbolic AI. In recent times AI has been eloquently described by Francois Chollet as "a general field that encompasses machine learning and deep learning, but that also includes many more approaches that don't involve any learning" [2]. Early AI systems consisted of a hardcoded set of rules crafted by their creators and seeing as no 'learning' is present cannot be considered as machine learning. This approach, known as *Symbolic AI,* was the main player in the field of AI from 1950 through to the late 1980s reaching peak popularity during the *expert system* boom of the 1980s.

Although Symbolic AI proved suitable in solving well-defined, logical problems, most noticeably playing chess, it turned out to be intractable in figuring out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, speech synthesis, and language translation [2]. A new approach arose to take Symbolic AI's place: *Machine Learning* (ML*)*.

ML arises from the questions*: Could a computer learn on its own rather than programmers crafting data-processing rules by hand? Could said computer automatically learn these rules by looking at data?*

These questions open the door to new type of programming paradigm. In classical programming, in this case Symbolic AI, a human input a set of rules (a program) and data to be processed according to these rules, and out comes the answer. With ML, humans input data as well as the answers expected from the data and out comes the rules. These rules can then be applied to new data to produce answers [2].

An ML system is trained rather than explicitly programmed where it is presented with many examples that are relevant to a specific task and through training, a statistical structure can be found that eventually allows the system to come up with rules for automating the task.

ML and statistics are tightly related, but they differ in several important ways. Unlike statistics, ML deal with large, complex datasets consisting of up to millions of examples for which classical statistics analysis would be impractical. As a result, ML, and especially Deep Learning (DL), exhibit comparatively little mathematical theory and can be seen as more engineering oriented due to its hand-on discipline in which ideas are proven empirically rather than theoretically.

As shown is Figure 1, DL is a specific subfield of ML which also provides a new insight on learning representations from data that emphasises learning successive *layers* of increasingly meaningful representations. The *deep* in DL does not refer to a deeper understanding achieved by the approach but rather, it illustrates the idea of learning successive layers of these representations [2].

In DL, these layered representations are learned via ANN based models but before we discuss ANNs in further detail we need to understand datasets, dataset features, dataset labels and different types learning methods.

3 Types of Learning

Within ML and DL there are three approaches to training ANNs: supervised, unsupervised and semi-supervised. If the dataset is completely labelled, the problem is called supervised learning. In unsupervised learning problems, the task is to find patterns and structures within unlabelled data. When a portion of the data is labelled by experts, the problem is called semi-supervised learning.

To make the idea of labelled data more concreate, we will use NSL-KDD dataset [3, 4, 5, 6, 7]. This dataset can be considered a '*hello world*' dataset used to illustrate ML and DL models for Intrusion Detection Systems. In the NSL-KDD dataset there are 41 features which are derived from TCP/IP connections and a label for each of the records making 42 attributes. The label assigned to each record denotes an attack type or a normal record. Table 1 shows the first 20 rows of the NSL-KDD training dataset. The last column of the table is the label categorised as normal class or different attack classes.

	duration	protocol_type	service	flag	src_bytes	...	dst_host_rerror_rate	dst_host_srv_rerror_rate	label
0	0	tcp	ftp_data	SF	491	...	0.05	0.00	normal
1	0	udp	other	SF	146	...	0.00	0.00	normal
2	0	tcp	private	S0	0	...	0.00	0.00	neptune
3	0	tcp	http	SF	232	...	0.00	0.01	normal
4	0	tcp	http	SF	199	...	0.00	0.00	normal
5	0	tcp	private	REJ	0	...	1.00	1.00	neptune
6	0	tcp	private	S0	0	...	0.00	0.00	neptune
7	0	tcp	private	S0	0	...	0.00	0.00	neptune
8	0	tcp	remote_job	S0	0	...	0.00	0.00	neptune
9	0	tcp	private	S0	0	...	0.00	0.00	neptune
10	0	tcp	private	REJ	0	...	1.00	1.00	neptune
11	0	tcp	private	S0	0	...	0.00	0.00	neptune
12	0	tcp	http	SF	287	...	0.00	0.00	normal
13	0	tcp	ftp_data	SF	334	...	0.00	0.00	warezclient
14	0	tcp	name	S0	0	...	0.00	0.00	neptune
15	0	tcp	netbios_ns	S0	0	...	0.00	0.00	neptune
16	0	tcp	http	SF	300	...	0.00	0.00	normal
17	0	icmp	eco_i	SF	18	...	0.00	0.00	ipsweep
18	0	tcp	http	SF	233	...	0.02	0.00	normal
19	0	tcp	http	SF	343	...	0.00	0.00	normal

Table 1: First 20 rows of NSL-KDD training dataset only few features are shown. Note NSL-KDD is an old dataset not representing current attacks, however it's still used by researchers to present models.

From the NSL-KDD dataset we will only reference the *training* dataset and the *testing* dataset. The training dataset contains 125,973 rows of data and the test dataset contains 22,544 rows.

The datasets are also imbalanced and must be pre-processed before use. An example of this imbalance is the number of *service* categories and classes of *labels* in each dataset; the training dataset contains 70 *service* categories, while test dataset contains 64 and there are 23 classes of *labels* in the training set, while the test dataset has 38.

4 Artificial Neural Networks

A neuron, the basic building block of ANN, was first introduced in the 1950s [1].

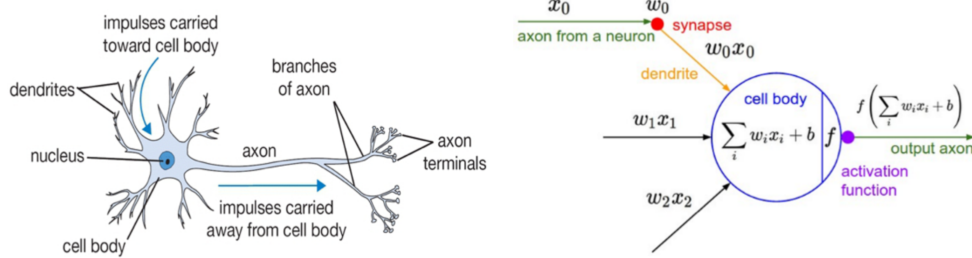

Figure 2 The structure of biological neuron and artificial neuron [8]

While inspired by the human brain and how biological neurons interact with each other, ANNs are not meant to be realistic models of the brain. Instead, they are an inspiration allowing us draw parallels between a very basic model of the brain and how we can mimic some of this behaviour through ANNs.

It works by taking feature inputs $\{x_0, x_1, x_2, ..., x_n\}$ and producing a single weighted output, $\hat{y}$, by multiplying the inputs with weights $\{w_0, w_1, w_2, ..., w_n\}$ and adding a bias value, b, and then applying a function, *f(·)*, to produce y. However, a neuron can only have a single output, making impossible to extend the model to work on classification tasks with multiple categories. This issue is solved by having multiple neurons in a layer, such that all these neurons receive the same input and each one is responsible for one output of the function. In fact, ANNs are nothing more than layers of neurons. Figure 3 shows a 2-layer model with an output layer predicting classes.

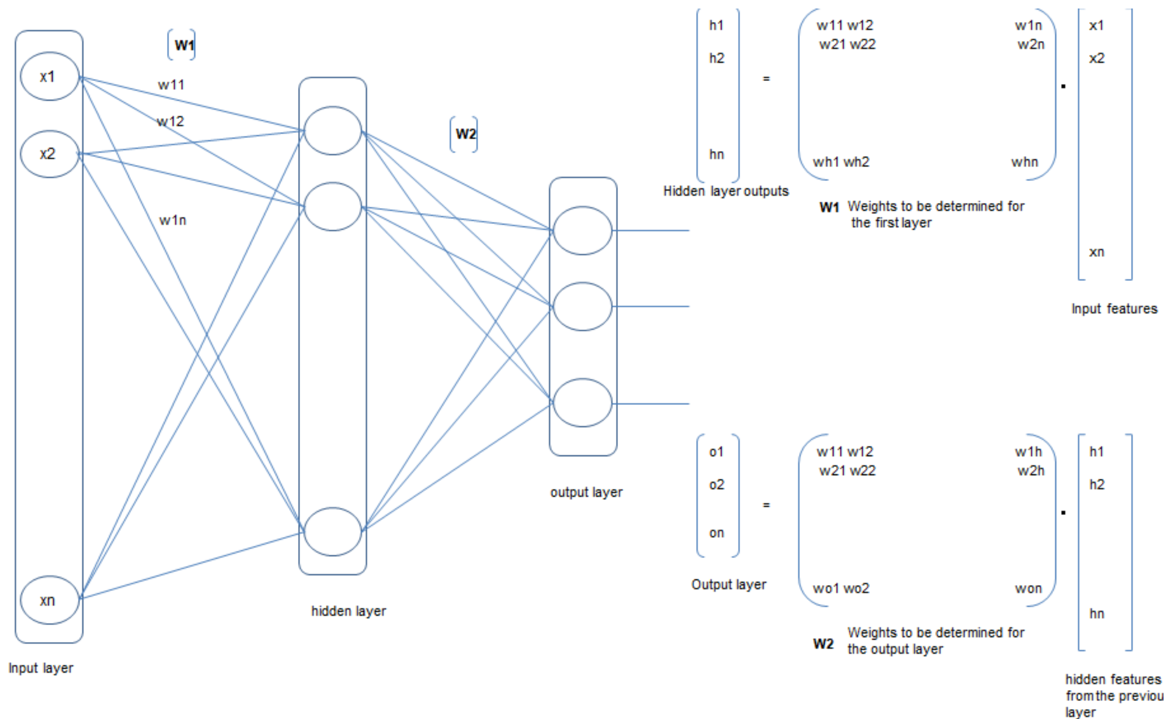

Figure 3: layer model with output layer predicting classes.

The problem is how do we find the weights and bias values at each layer to give us the desired accuracy. We know the input features, $\{x_0, x_1, x_2, ..., x_n\}$ and the labels, $\{y_0, y_1, y_2, ..., y_n\}$ from our training dataset. In the case of NSL-KDD dataset we have 41 features, 23 different labels and in total of 125,973 rows of training data. So, we want to train a neural network using the 125,973 rows of training data.

Once the model is trained and deployed we can expect the model to output the probabilities of each of the 23 labels. If the model is fed with input features which correspond to features of normal data packet we should get a high probability of that data packet being normal. If the data packet has the features of an attack, then model

should predict which particular type of attack. The output class with the highest probability would indicate the type of attack.

But how do we go about acquiring the ideal weight matrix, W, and bias vector, b, that will obtain high classification accuracy?

Do we randomly initialise these weights, evaluating and repeating over and over until at some point we land on a set of parameters that obtains reasonable classification? We could, but this is not practical given that modern neuron networks have parameters in the order of tens of millions. Instead of relying on pure randomness, we need to define an optimisation algorithm that allows us to literally improve W and b. The most common algorithm used to train neural networks is *gradient descent*. Gradient descent has many variants, but in each case the idea is the same: iteratively evaluate your parameters, compute your loss, and then take a small step in the direction that will minimise your loss. The interaction between these elements is illustrated in Figure 4.

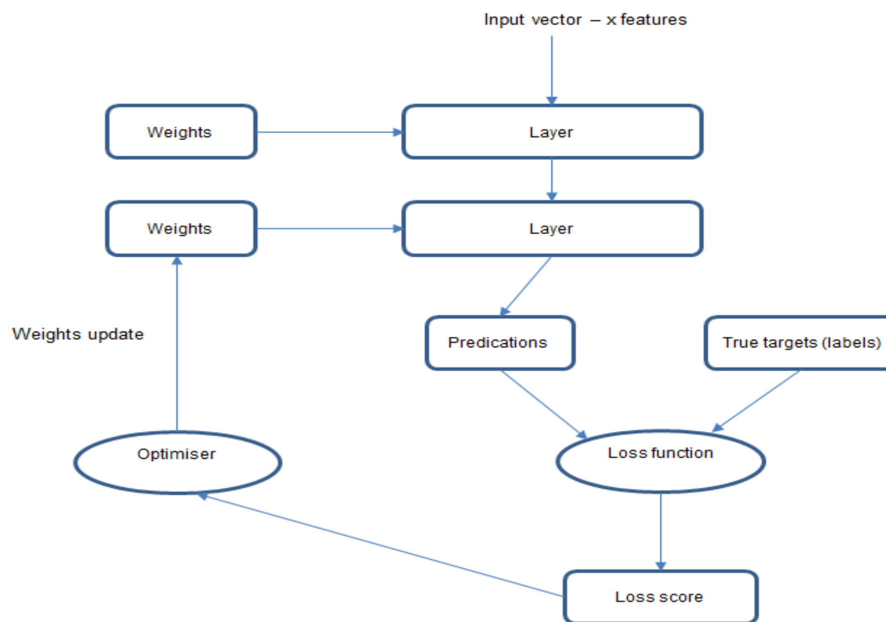

Figure 4 Interaction between the network layers, loss function and optimiser [2].

Taking the small step in the direction that will minimise the loss is controlled by the *Backpropagation* algorithm. At the heart of *Backpropagation* algorithm is an expression for the partial derivate, the rate of change of loss with respect to each of the weights and bias within the network. This expression tells us how quickly the loss would change when the weights and biases are updated thus providing detailed insights into how altering the weights and biases would affect the overall behaviour of the network.

5 Training Artificial Neural Networks

There are four main steps to constructing a neural network:

1) Gather and pre-process your dataset

The first component of building a neural network is to gather our initial dataset. This is the most crucial and time-consuming task, particularly for cybersecurity use cases as we will see when discussing Intrusion Detection Systems and Malware detection. For computer vision and other domains, it is a lot easier to 'have a feel' for the data features. For example, in image classification we have an image consisting of width, height and colour channel (RGB) pixels as well as the category of image. For speech to text we have the 1D speech signal and associated labels. For machine translation and language modelling we have set of documents.

For cybersecurity use cases there is a higher degree of diversity than most people intuitively expect, which in turn leads to misconceptions about what is realistically achievable. Even with a single network, the network's most basic characteristics such as bandwidth, duration of connections, and application mix can exhibit immense variability. So, deciding on which feature(s) to capture is a difficult and challenging task for cybersecurity.

For researchers arguably the most significant challenge cybersecurity faces are the lack of appropriate and up to date public datasets. In other domains large datasets are readily available, such as ImageNet for image classification.

Dataset must be analysed and pre-processed before it is ready to be used. Looking at the NSL-KDD dataset there are 3 types of features **`int64`**, **`object`** and **`float64`**:

```
duration int64, protocol_type object, service object, flag object, src_bytes int64, serror_rate float64, label object
```

All features are numerical, except `protocol_type` *(has 3 categories:* `udp, tcp, icmp`*),* `service` *(has 70 categories)* and `flag` *(has 11 categories)* which are string type. Most ML algorithms prefer to work with numerical data, so these features will have to be converted to numbers. A common method is to create one binary attribute per category: one attribute equal to '1' when the category is `tcp` (and '0' otherwise), another attribute equal to '1' when the category is `udp` (and 0 otherwise), and so on. This is called one-hot encoding, because one attribute will be equal to '1' (hot), while the otherwise will be '0' (cold). So for this case 3 new features are added `protocol_type_icmp`, `protocol_type_tcp` and `protocol_type_udp` while the original feature `protocol_type` is dropped. The `service` and the `flag` features are handled in the same way resulting in 123 features instead of 41 features in the un-processed dataset.

Feature scaling, a common requirement of ML methods, is also applied to prevent features with large values from significantly impacting the model accuracy. For each feature, compute the mean, subtract this mean value from the feature value, and divide the result by their standard deviation. This is a standard normalisation technique for many datasets.

There is also discrepancy in the `service` feature; in the training set there are 70 `service` categories while the test set only contains 64. For the `label` feature the training set has 23 categories while the test set has 38 categories. Therefore, the datasets must be balanced by either including the missing categories or deleting them.

Another option is to re-group the label categories in to 5 categories `Normal`, `DOS`, `Probe`, `Remote to Local` and `User to Root`.

`DoS` `back, land, neptune, pod, smurf, teardrop`

`Probe` `satan, ipsweep, nmap, portsweep`

`R2L` `guess_passwd, ftp_write, imap, phf, multihop, warezmaster, warezclient, spy`

`U2R` `buffer_overflow, loadmodule, rootkit, perl`

As you can see, pre-processing and feature engineering is a very involved task and considerable time needs to be devoted before the data is ready to be used in developing classifiers.

2) Split Your Dataset

Now that we have our initial dataset, we need to split it into two parts, *training set* and *testing test*. A *training set* is used by the neural network to *learn* what a category looks like by making predictions on the input data and then correcting itself when predictions are wrong. After the neural network has been trained, we can evaluate the performance on a *testing set*.

It is extremely important that the training set and testing test set are independent of each other and do not overlap. If you use your testing set as part of your training data, then your neural network has an unfair advantage since it has already seen the testing examples before and learned from them. As a result, the evaluation would be misleading. Instead, you must keep this testing set entirely separate from your training process and use it only to evaluate your network.

Common percentage split sizes for training and testing sets include 70/30 and 80/20.

Neural networks have a number of *hyperparameters* used by the backprogation algorithm (learning rate, decay etc) that need to be tuned. In practice, we need to test these hyperparameters and identify the set of parameters that work the best. For this we create a third data split called the validation set. This set of data comes from the training set so we can tune our hyperparameters. We normally allocate roughly 10%-20% of the training data for validation.

3) Train Your Network

Before we can train our neural network, we need to decide its architecture. This is number of layers, types of layers, number of parameters at each layer etc. Designing neural network architecture is an iterative process and not an exact science and comes with experience.

A number of iterations of the neural network will be required to arrive at the desired training and validation accuracy. If the desired accuracy cannot be achieved, dataset features may need to be reassessed.

4) Evaluate

The next step is to evaluate our trained network. For each of the rows in our testing set, we present them to the network and ask it to predict what it thinks the attack label is.

Finally, the model predictions are compared to the *ground-truth* labels from our testing set. The ground-truth labels represent what the attack category actually is. From there, we can compute the number of predictions our classifier got correct and compute aggregate report such as precision and recall, which are used to quantify the performance of our network as a whole (sometimes accomplished with an F1-score).

Model deployment

Once a neural network has been trained it can finally be used to make predictions on previously unseen data points. By deploy the model operationally it can be utilised to make predictions continuously from an input data stream. The data input stream must be collected and be pre-processed in the same way as the training set was. The topology of the finalised neural network model and the final set of weights is all that you need when saving the model. Predictions are made by performing a forward-pass of the trained network model on a given input. An output prediction is then computed.

6. AI – Cybersecurity use cases

There is tremendous interest in the cybersecurity arena with regards to use of AI, particularly in areas such as Network Intrusion Detection (NID), Malware Detection, Phishing Detection, classifying encrypted traffic and Security Surveillance.

In this section we will discuss each one of these use cases and highlight the challenges we are facing. For each of the use cases we will introduce new type of ANN. We start with Autoencoders (AEs) which are being used in NID systems. Convolution Neural Networks (CNNs) will be introduced when discussing Malware Detection using a novel approach, where we turn malware files into images then apply a CNN. The section on Phishing Detection will introduce Natural Language Processing (NLP) for processing text documents. We finish the section with a discussion on Security Surveillance applications, which perhaps is most mature of the use cases we will discuss.

6.1. Network Intrusion Detection

The idea of supervised training in NID has already been discussed briefly; if a good labelled dataset can be engineered then developing a neural network model for detecting anomalies in the network traffic is achievable.

However, there is a major drawback with this approach: *how do we detect unknown anomalies for which the neural network was not trained for?* So, if new attack is detected the neural network would have to be re-trained and re-deployed. This in essence takes us back to a different kind of signature-based deployment however, if ANNs can be demonstrated to work better than the more traditional methods, deploying a new ANN as required shouldn't be an issue.

Many challenging obstacles surface when striving to develop effective and flexible NIDs for unknown future attacks. The most notable challenge being suitable feature selection from network traffic dataset for anomaly detection. The continuously changing and evolving nature of attack scenarios means that features selected for one class of attack may not work well for other classes of attacks [9].

Recently, use of AE methods [9, 10, 11, 12, 13, 14, 15] have come into play for developing flexible NIDs. These methods aim to learn a good feature representation from a large amount of unlabelled data.

Autoencoding is a data compression and decompression algorithm implemented with an ANN. Since it is an unsupervised form of a learning algorithm, we know that only unlabelled data is required. Figure 5 illustrates the architecture for a simple AE; a compressed version of the input is generated by forcing it through a bottleneck (encoding layer), a layer or layers with a smaller width compared to the original input. This process allows the network to learn a compressed representation of the input. To reconstruct (decompress) the input we reverse the process using decoding layers. Backpropagation is used to both create the representation of the input in the intermediate layer(s) and recreate the input as the output from the representation.

The reconstruction (decompressed output) will be degraded with respect to the original input, thus AEs are considered 'lossy'. Autoencoding is data-specific, meaning only data similar to that it has been trained on is properly compressible (able to learn a compressed representation). For example, an AE trained on features which are considered to be normal would perform badly on features considered an attack. When that happens, by observing the result it can be deduced with high probability that the input is an attack.

To illustrate how AE can be designed for NID systems we can take all of the rows which have the label *normal* from the NSL-KDD training set and feed these rows to our neural network during training and try to re-construct the input features. When input features can be re-constructed with reasonable accuracy then the training phase is complete.

AEs trained on $\mathbf{X}=\{x_0, x_1, x_2, ..., x_n\}$ (training set) have the capability to reconstruct unseen instances that have a similar data distribution to $\mathbf{X}$. If an instance does not belong to the representations learned from X, then we expect the reconstruction to have a high error. The reconstruction error of an instance x for a given AE, can be computed by taking the root mean squared error (RMSE) between the input x and the reconstructed output. We can detect anomalies when the error between input features and the re-constructed features is above a certain threshold.

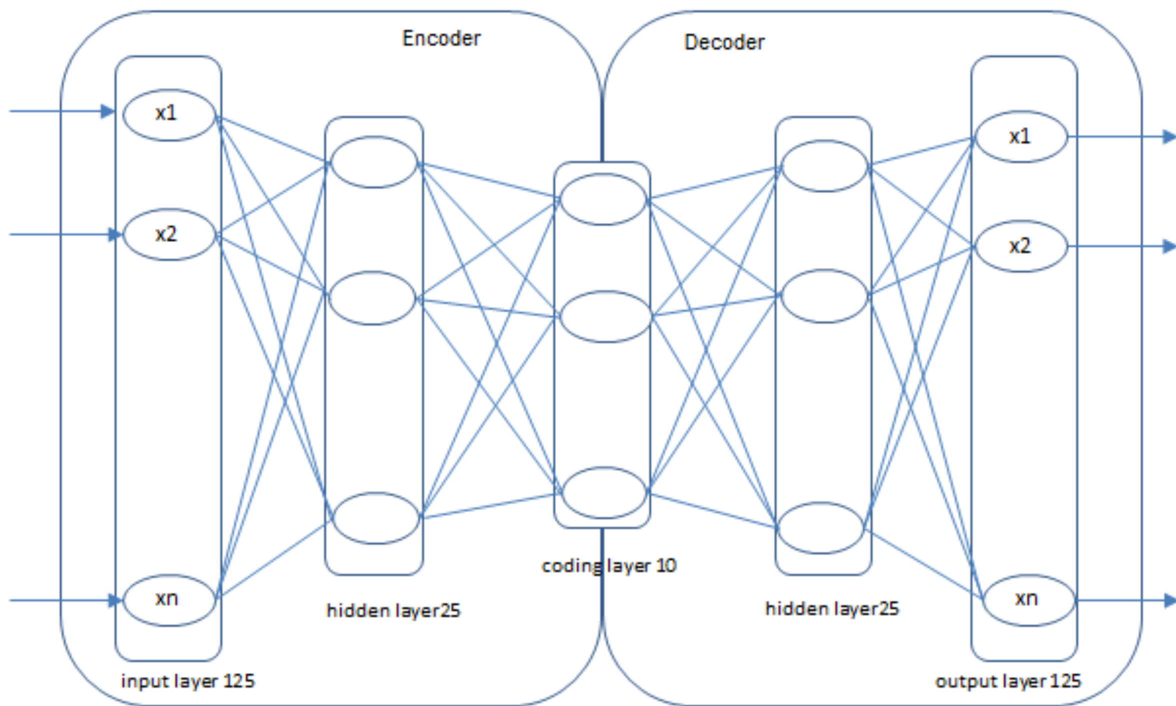

Figure 5 Multi-layer Auto-Encoder.

Just like other neural networks we have discussed, AEs can have multiple hidden layers. In this case they are called Stacked AEs. Stacked AEs tend to have a symmetrical architecture either side of the central (coding) layer. For the NSL-KDD dataset we 125 input features, followed by a hidden layer with 25 neurons, then a central hidden layer of 10 neurons, then another hidden layer with 25 neurons, and an output layer with 125 neurons. Table below shows the parameters to be trained for each layer.

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 25)	3150
dense_2 (Dense)	(None, 10)	260
dense_3 (Dense)	(None, 25)	275
dense_4 (Dense)	(None, 125)	3250

Total params: 6,935
Trainable params: 6,935

In the scenario of NID one question remains: *what is normal*? The statistics of network traffic in the morning is likely to be different in middle of the day for any given system and company. Network traffic statistics may vary day to day as different applications are run on different days or months. These are points of consideration that need to be addressed when designing an AE for NID.

6.2 Malware Detection

Detection of malicious software is done mainly with heuristic and signature-based methods that struggle to keep up with malware evolution. Heuristics based approaches are based on rules determined by experts, relying on dynamic and static analysis of malicious behaviour and thus, it can deal with unknown malware, but it also generates greater amounts of false positives than signature-based methods. As a result, Anti-Virus vendors attempted to use hybrid analysis approaches by using both signatures-based and heuristics-based methods to detect malware [16, 17, 18, 19, 20, 21, 22, 23].

In recent years, the possibility of ML methods to detect malware has been increasing, with these approaches holding the promise of achieving high detection rates.

In ML approaches, most of the effort goes into gathering malware files and feature engineering (deciding which features of the malware files to use in training the neural network), again not an easy task. Features can be selected based on *Static analysis*, *Dynamic analysis* and *Hybrid analysis*.

Hybrid analysis features are obtained by combining both techniques. Static analysis features include: *byte code n-gram features*, *opcode n-gram features*, *portable executables* and *string features*. While the dynamic analysis features include *function based features*, *API calls, system calls* and *information flow tracking*.

To train a reasonably accurate malware classifier one needs a dataset consisting of tens of thousands of malware files. In 2015 Microsoft hosted a competition on Kaggle with the goal of classifying malware into their respective families based on their content and characteristics. Microsoft provided a dataset with a total of 21741 samples, 10868 for training and 10873 for testing (almost half a terabyte when uncompressed) [16, 24]. This dataset represented 9 different malware families.

There are over 50 papers citing this dataset, each using various type of features and ML techniques, however as yet there no evidence of any comparative study conducted on these papers and no guidance available as to which features of the malware are effective in their detection.

To overcome feature selection problem, more radical methods of malware detection have been suggested. One idea is to convert malware files into greyscale images and then train CNNs to classify them. This idea is based on the observation that images of different malware samples from the same family appear to be similar, while images of malware samples belonging to a different family are distinctive from each other. Moreover, if old malware is re-used to create new malware binaries the resulting one would be very similar visually.

A given malware binary file is read as a vector of 8-bits unsigned integers and then organised into 2D array which can then be visualised as a greyscale image. The width of the image is fixed, and the height is allowed to vary depending on the file size.

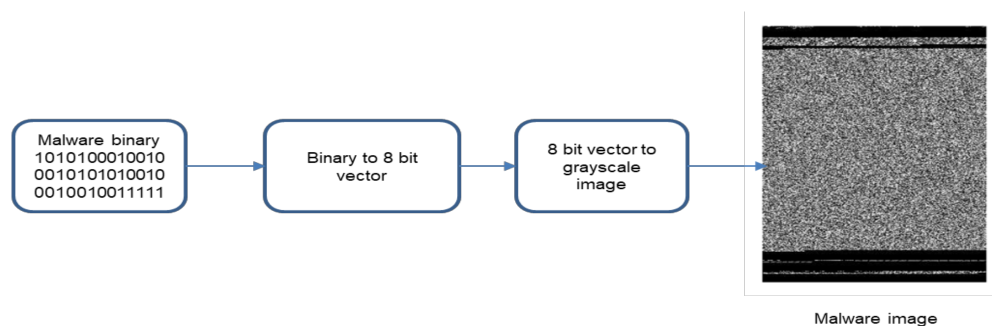

Figure 6 Visualising Malware as a grey-scale image

Each of the malware files in the training set and testing set are converted to images. Then these images are used to train a CNN classifier to make predictions.

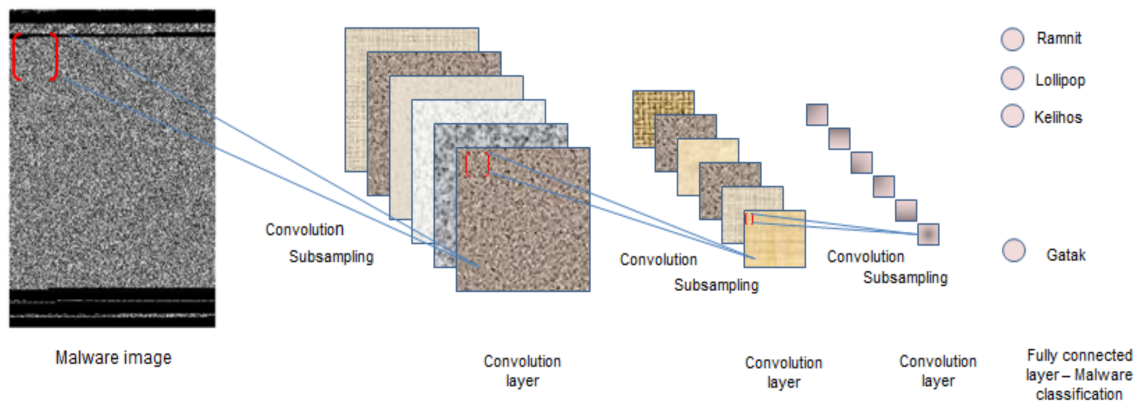

Figure 7 Convolutional Neural Network for Malware Classification

The action of convolution in a CNN is the process of adding each element of an image within a specific window size weighted by a filter of the same size. Consider a 128x128 malware image, a CNN can efficiently scan it using say, a 3×3 window (2D filter). The 3×3 window moves from left to right, top to bottom through the image. Element-wise multiplication of the 3x3 region of the image and the 3x3 filter takes place. The elements of the resulting 3x3 matrix are added together to produce a pixel value for that 3x3 area of the image. This process continues until all pixels in the image have been covered thus creating a new image, illustrated in Figure 7.

How 'quickly' the filter moves from left to right is denoted by its stride length e.g. stride length of 1 means the 3×3 window slides by 1 pixel at a time until it has covered the entire image.

This process (convolution) happens in the convolution layer of the neural network. A typical CNN has multiple convolution layers. Each convolutional layer typically generates many alternate representations, so the weight matrix is a tensor of 3×3×n, where n is the number of convolution filters.

At each layer in a CNN a different representation (learnt patterns) of the malware is learned by applying different sets of filters at each layer (typically 64 or 128 filters of size 3x3). During training, a CNN automatically learns the values for these filters through the backpropagation algorithm, described in section 4 Artificial Neural Networks.

The last layer in a CNN uses the learned representation to make predictions regarding the classification of the malware. For the Microsoft challenge the output layer would have 9 neurons each representing the probabilities of the result belonging to each of the 9 malware families.

For a more detailed description of convolution layers see [25].

6.3 Phishing detection

There are a number of approaches on deciding which types of features to use when developing neural networks for phishing detection. Various website characteristics can be used to aid in distinguishing phishing websites from legitimate websites. Such characteristics include: long URLs, IP address in URLs, and request access to additional URLs in which these characteristics are the indications of being phishing websites [26]. Another approach is to just use the content of the email.

Phishing detection based on the content of the emails relies on Natural Language Processing (NLP). The goal of NLP is to process text for document classification, sentiment analysis such as classifying sentiment of email as spam or ham, sequence to sequence learning such as decoding English into French, language modelling to predict the next word in a sentence.

When working with text it is common place to work at word level rather than a sequence of characters or a sequence or words.

Like all other neural networks, ML or DL models do not take inputs as raw text. They work best with numeric vectors. The process of transforming text into these numeric vectors is *Vectorising*.

Collectively, the different components into which you can break down a piece of text (words and characters) are called *tokens*. *Tokenisation* is the term given when breaking text into these tokens. All text–vectorisation processes consists of applying some form of tokenisation technique and a method for associating the vectors with generated tokens (there are multiple ways to associate a vector with a token.). These vectors are then organised into sequence tensors and are given as inputs to neural networks.

A popular and powerful system to associate a vector with a given word is the use of *word embeddings*. Vectors obtained through one-hot encoding are binary and mostly made of zeros (sparse) and can be very high-dimensional due to having the same dimensionality as the number of words in the vocabulary used. However, word embeddings have low dimensionality by utilising floating-point vectors which by their nature are dense and thus better suited than sparse vectors.

Feature selection for classifying a phishing email's content does not pose an issue if it is just sequence of words in text files. The collection of text files labelled as *ham* or *spam* is where most of the effort will be spent.

Public phishing datasets are available, but the quality and size of these datasets vary from dataset to dataset. For purpose of our discussion we will use dataset available from [27].

The SMS spam collection contains 4827 legitimate SMS messages and 747 spam messages, therefore heavily imbalanced. The dataset is presented in a single file where each line corresponds to a single data sample; the first word is the label and the rest is the SMS content. The first line is as follows:

ham Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amore wat...

After tokenisation the line becomes:

[49, 471, 4435, 842, 755, 658, 64, 8, 1327, 88, 123, 351, 1328, 148, 2996, 1329, 67, 58, 4436, 144]

Tokenisation produces word index, a dictionary of words with their uniquely assigned integers. A snippet of this word dictionary looks like:

{'i': 1, 'to': 2, …, 'go': 49, …, 'until': 471, ..., 'jurong': 842, …, 'point': 4435, ..., 'bitching': 9009}

By viewing this word dictionary, it can be seen that the vocabulary size is 9009.

Because we have little training data available, we cannot use our dataset to learn appropriate word embeddings of our vocabulary. Instead of explicitly learning the word embedding jointly with the current problem we want solve, we can instead load embedding vectors from a precomputed embedding space that is highly structured and exhibits useful properties that captures generic aspects of language structure [28].

Word2vec and *GloVe* (Global Vectors for Word Representation) are examples of standard databases providing pre-computed word-embeddings that can be downloaded and subsequently used in a neural network's embedding layer.

The smallest pre-computed *GloVe* model available [29] is an 822 MB zip file with 4 different models (50, 100, 200 and 300-dimensional vectors) trained on Wikipedia data with a total of 6 billion tokens and a 400,000-word vocabulary.

The following snippet shows the vector (of length 100) for the word '*the*':

[-0.038194 -0.24487 0.72812 -0.39961 0.083172 0.043953 -0.39141 0.3344 -0.57545 0.087459 --0.073438 -0.28312 0.37104 -0.25217 0.016215 -0.017099 -0.38984 0.87424 -0.72569 -0.51058 -0.52028 -0.1459 0.8278 0.27062]

To build an embedding matrix for our embedding layer we take our vocabulary built during tokenisation and combine it with the *GloVe* model. The matrix shape would be 9009x100, maximum number of words in vocabulary by the embedding dimension, where each entry, *i,* contains the embedding vector for the word on index, *i,* in the reference word index (built during tokenisation). This matrix allows the captures of information between words and how they are used in sentences.

By definition, sentences consist of a sequence of words. Sequence data is best handled by a special kind of neural network layer built from Long Short-Term Memory (LSTM) units. Unlike other types of neural network layers LSTM have a sense of memory and are much more complex than other type of layers

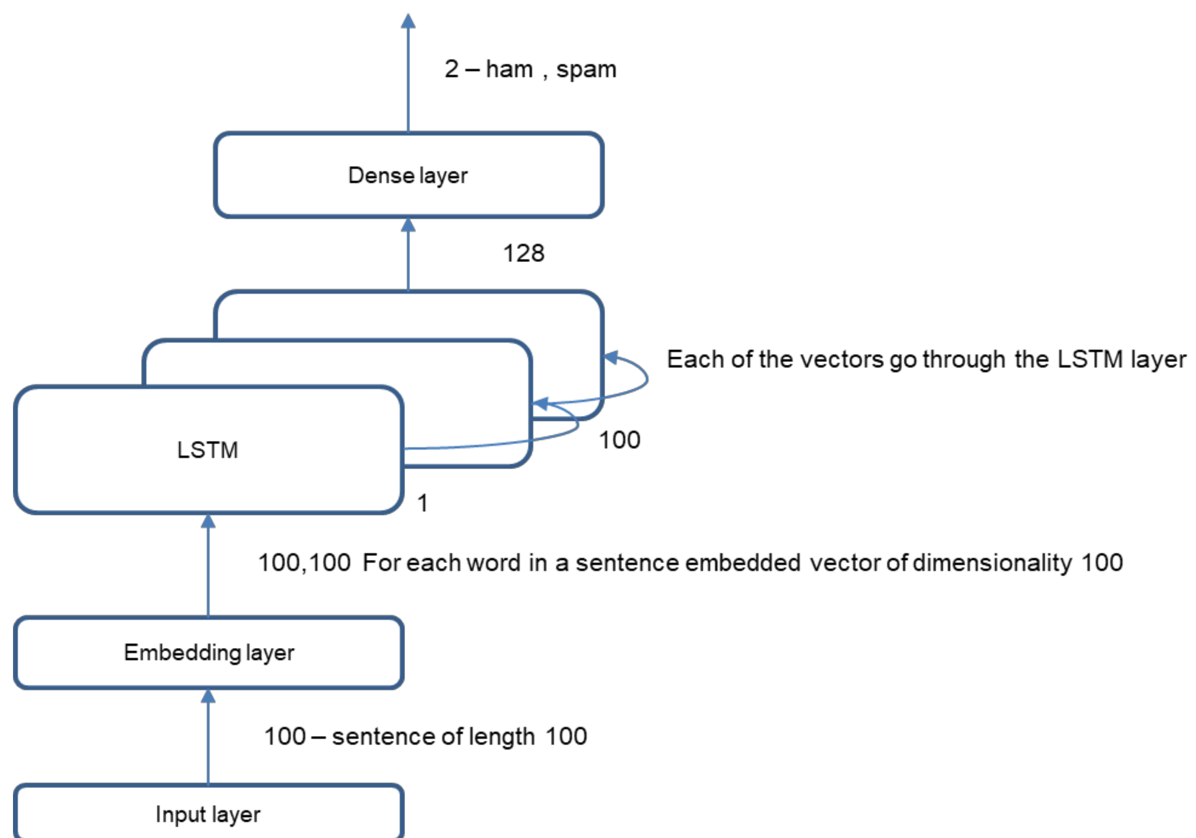

Figure 8 Simple model to classify emails.

There are other Cybersecurity use cases where NLP can be used effectively, for example a Security Operation Centre generates enormous amounts of logs which needs to be interpreted by security analysts. The process can be automated by using an ANN trained to classify different types of logs and alert the security analysts as required. It's only a matter of creating datasets with different types of security logs!

6.4 Security Surveillance

Security surveillance is a repetitive and mundane task and due to this nature may cause performance dips for us humans but by allowing technology, specifically AI based, to carry out this surveillance, we could then focus on taking actions on more pressing tasks that require human input.

Security surveillance is the most mature of the cybersecurity use cases discussed so far. This shouldn't be a surprise since it relies on image classification, object detection and image segmentation, areas of which have been most successful in deep learning.

The field of Computer Vision can be divided into three subgroups [30]:

Image Classification: Predict the type or class of an object in an image.

– **Input:** An image with a single object, such as a photograph.

– **Output:** A class label (e.g. one or more integers that are mapped to class labels).

Object Localisation: Locate the presence of objects in an image and indicate their location with a bounding box.

– **Input**: An image with one or more objects, such as a photograph.

– **Output**: One or more bounding boxes (e.g. defined by a point, width, and height).

Object Detection: Locate the presence of objects with a bounding box and the classes of the located objects.

– **Input:** An image with one or more objects, such as a photograph.

– **Output:** One or more bounding boxes (e.g. defined by a point, width, and height), and a class label for each bounding box.

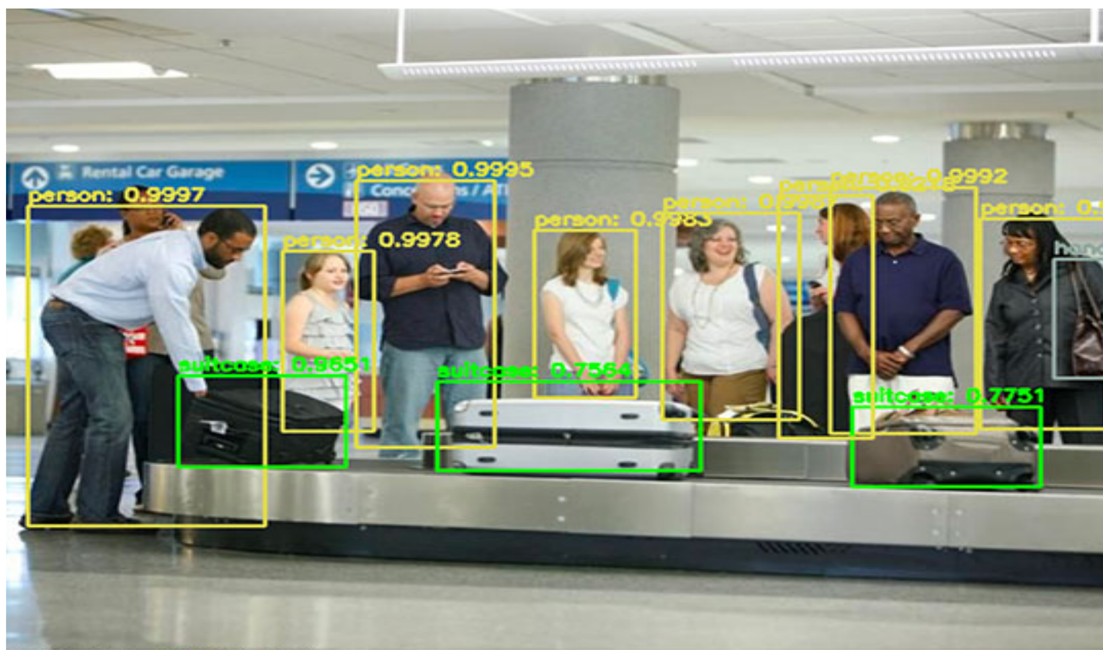

Figure 9 Image classification and object detection [31]

The success of this domain can be attributed to the availability of large public datasets such as the ImageNet dataset which contains approximately 1.2 million training images, 50,000 validation images, and 100,000 testing images. The goal of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [32] is to develop and train a model that can classify an image into one of 1000 separate classes which represent objects we encounter in our everyday lives, ranging from different vehicle types, breed of dog or cat to a multitude of household objects and a vast array of many more.

When it comes to image classification, the ILSVRC challenge is the de facto standard for computer vision classification models and the winning techniques of this challenge has been dominated by deep learning methods since 2012. Table 2 shows the list of image classification models and their contributions to deep learning models.

Network Architecture	Year	Contribution
AlexNet	2012	The work that perhaps could be credited with sparking renewed interest in neural networks.
VGGNet	2014	The first important difference leading to standard practices of using a large number of small filters. Specifically, filters of size 3x3 and 1X1 with stride of 1. A number of variants of this architecture were developed VGG-16 and VGG-19 (16 and 19 denoting the number of layers).
GoogleNet - Inception	2014	The key innovation of the GoogleNet models are the *Inception* modules. This is a block of parallel convolution layers with different filter sized filters (1x1, 3x3, 5x5) the results of which are then concatenated.
ResNet	2015	This model contains 152 layers and introduces the idea of residual blocks that make use of skip connections. These are connections in the network architecture where the input is not weighted and is passed on to a deeper layer.

Table 2 Image classification models

The above networks' accuracies improve upon one another however, this increased complexity and deeper architectures have prevented using these models on computationally limited platforms such as mobile and embedded systems. To overcome these limitations, new and efficient network architectures have been developed which can be used in these mobile and embedded computer vision applications.

MobileNets [33] are a family of mobile-first computer vision models, designed to maximise accuracy while operating under the constrained resources of an on-device or embedded application. In terms of performance, MobileNet (4 million parameters) is roughly 3 times faster than Inception (24 million parameters) and 10 times faster than VGG-16 (138 million parameters). On a modern smartphone, MobileNet can run at 20 Frames Per Second with an ImageNet top-1 accuracy of 70.6% [33] which is less than the more complex networks.

Image classification models are also incredibly useful for transfer learning. Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task.

"Transfer learning is the improvement of learning in a new task through the transfer of knowledge from a related task that has already been learned." - Francois Chollet [2]

Some of these models are *base networks* for the object detection models and face recognition models.

Network Architecture	Year	Base Network	Type	Contribution
Faster R-CNN (Regional CNN)	2013 2015 2015	VGG-16, Inception, ResNet	Two-Stage	This is a family of object detectors whose accuracy, speed and model size improved over 3 iterations from R-CNN, Fast R-CNN to Faster R-CNN. Two stage detectors, such as R-CNN are more accurate than one-stage detectors such as SSD and YOLO.
SSD (Single Shot Detectors)	2015	VGG-16, ResNet MobileNet	Single Stage	Overcomes the speed restrictions of the Faster R-CNN detectors but there is small drop in accuracy.
YOLO-v3 (You Only Look Once)	2015 2018	Darknet-19 Darknet-53	Single Stage	This is a family of object detectors whose accuracy, speed and model size improved over 3 iterations from YOLO, YOLO-v2 and YOLO-v3
RetinaNet	2017	ResNet	Single Stage	RetinaNet can be considered a cousin of both the R-CNN family and the SSD family of object detectors. RetinaNet detectors could match the speed of existing one-stage detectors while obtaining superior object detection accuracy over two-stage R-CNN inspired detectors. RetinaNet therefore seeks to obtain the best of both worlds: speed and accuracy.

Table 3 Object detection models

Common datasets used to train object detection models include Microsoft COCO, Pascal VOC2012 and KITTI. The VOC2012 dataset consists of 20 classes, including objects most commonly captured on traffic cameras like bus, car, bicycles, motorcycles and person. The dataset has been split into 50% for training/validation and 50% for testing. The COCO dataset meanwhile consists of 80 classes.

To train your own deep learning objector the first step is to create your dataset. Creating a dataset for object detection is also time-consuming task. The first task is to collect images containing your desired objects, how many images to collect will depend on the training method intended to be used.

The next step is to label each of the objects in each of the images, this can be done with annotation tools which allows the user to load the image into the application, then the user can draw the bounding boxes around the objects in the image. Finally, the details are exported into an XML file.

To illustrate these steps, we will use the dataset from [34]. The dataset itself consists of 3000 images containing handguns and 3000 annotations files describing the bounding boxes. First image and its annotation file are shown below.

Figure 10 Image from weapons dataset (armas_1.jpg)

The weapons annoatations are stored in PASCAL VOC format, a very popular format used when annotating images for object detection. A sample of one the weapon's XML files can be seen below in Figure 11.

```
<annotation>
  <folder>Definitiva</folder>
  <filename>armas (1)</filename>
  <path>..\armas (1).jpg</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>240</width>                  Image size (3 channels/depth for RGB)
    <height>145</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>pistol</name>                 Class name
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>                            Bounding box
      <xmin>3</xmin>
      <ymin>1</ymin>
      <xmax>128</xmax>
      <ymax>100</ymax>
    </bndbox>
  </object>
  <object>
    <name>pistol</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>                            Bounding box
      <xmin>123</xmin>
      <ymin>47</ymin>
      <xmax>238</xmax>
      <ymax>145</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 11 Annotation file for arams_1.jpg

An implementation of RetinaNet can be installed from [35]. Once installed, the model to detect hand pistols can be trained from the command line:

```
$ retinanet-train --batch-size 2 --steps 1309 --epochs 50 \
                  --weights weapons/resnet50_coco_best_v2.1.0.h5 \
                  --snapshot-path weapons/snapshots \
                  csv weapons/retinanet_train.csv weapons/retinanet_classes.csv
```

The '*retinanet_train.csv*' file and the '*retinanet_test.csv*' file are built from the images and annotation files and have the following format:

```
/home/weapon-detection/weapons/images/armas_1.jpg, 3, 1, 128, 100, pistol
/home/weapon-detection/weapons/images/armas_1.jpg, 123, 47, 238, 145, pistol
…
/home/weapon-detection/weapons/images/armas_3000.jpg, 123, 47, 238, 145, pistol
```

Once the model has been trained it must be converted to full inference model to be used for prediction.

```
$ retinanet-convert-model weapons/snapshots/resenet50_csv_50.h5 output.h5
```

The following command is used to evaluate the trained model using the '*retinanet_test.csv*' file.

```
$ retinanet-evaluate csv weapons/retinanet_test.csv weapons/retinanet_classes.csv output.h5
845 instances of class pistol with average precision: 0.8776
```

To achieve higher accuracy the dataset size needs to be increased.

To apply the weapon detector to an example image, the predict script can be called as follows:

```
$ python predict.py --model output.h5 --labels weapons/retinanet_classes.csv \
                    --image pulp-fiction.png \
                    --confidence 0.8
```

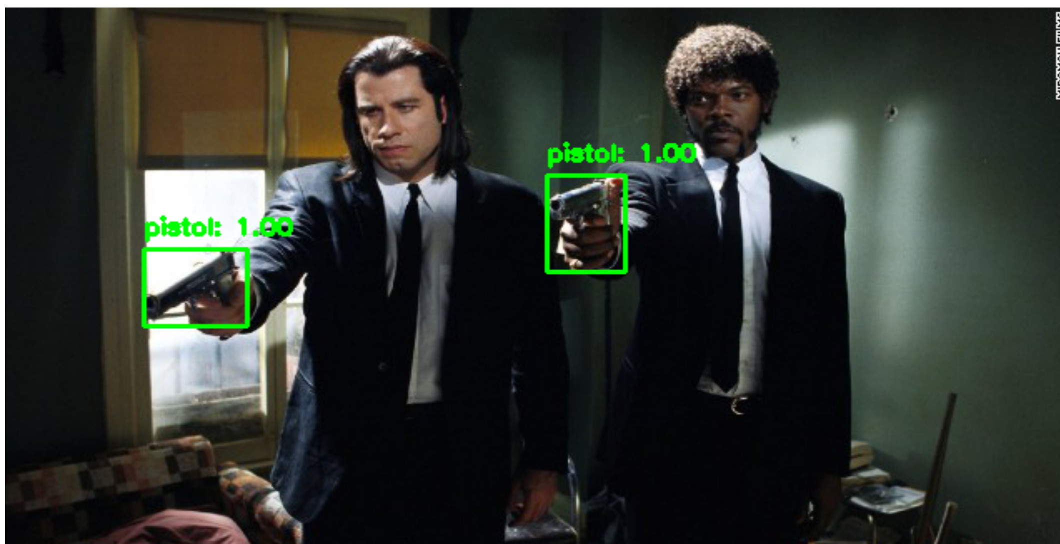

Figure 11 Detected **pistol** (Source Pulp Fiction motion picture)

The *predict* script can be integrated into surveillance applications where it can be called as each frame of video is captured and an alarm can then be raised if a pistol is detected in a video frame. To detect any other types of object such as unattended luggage, people wandering in a secure area, cars in a secure area and other types of weapons etc, we need to collect a large set of images annotate them by drawing a bounding box around each of the objects. Then using these image and annotations, create train and test csv files followed by calling the above commands.

There are other deep learning object detection API available such as [36, 37]. These API's generally follow the same principles as discussed above for RetinaNet.

A 2011 book on face recognition titled '*Handbook of Face Recognition*' [38] describes two modes for face recognition:

"A face recognition system is expected to identify faces present in images and videos automatically. It can operate in either or both of two modes: (1) face verification (or authentication), and (2) face identification (or recognition)." - Wheeler & Liu (Handbook of Face Recognition [38])

Face Verification/Authentication – A mapping of a given face to a known identity (is this the person?).

Face Identification/Recognition – A mapping for a face against a catalogue of known faces (who is this person?).

A helpful survey highlighting the history of facial recognition research over the past 30 years can be found in the paper *Deep Face Recognition: A Survey* [39]. This survey provides a "comprehensive review" that looks at early holistic models such as Eigenfaces, handcrafted feature detection, shallow learning and finally the recent state-of-the-art methods achieved through deep learning.

"The holistic approaches dominated the face recognition community in the 1990s. In the early 2000s, handcrafted local descriptors became popular, and the local feature learning approach were introduced in the late 2000s. shallow learning method performance steadily improves from around 60% to above 90%, while deep learning boosts the performance to 99.80% in just three years." - Deep Face Recognition: A Survey [39].

As mentioned previously, the re-emergence of neural networks for image classification can be credited to AlexNet (2012). This re-emergence led to a surge of research publications in 2014 and 2015 on deep learning based methods for face recognition. Over the subsequent 3 years capabilities of these approaches quickly achieved near-human-level performance and then exceeded when performing on a standard face recognition dataset. This astonishing rate of improvement in face recognition can be accredited to four landmark innovations: DeepFace, the DeepID series of systems, VGGFace, and FaceNet.

The VGGFace2 authors provided code for training their models, as well as pre-trained models for use by any user interested. These can be downloaded with standard deep learning frameworks [40].

FaceNet, a 2015 face recognition system developed by Google researchers, achieved state-of-the-art performance on a variety of face recognition benchmark datasets. Due to many open source, third-party implementations of FaceNet along with the availability of pre-trained models, this system can be used by anyone interested without much difficulty. Additionally, FaceNet can also be used for the task of extracting high-quality facial features (face embeddings) which itself can be used to train a separate face identification system.

6.4.1 Recognising Objects with Internet of Things (IoT)

Embedded AI based security surveillance systems can now be developed using Internet of Things devices such as the Raspberry Pi (RPi), Google Coral USB accelerator [41], Intel Movidius Neural Compute Stick [42] and the NVIDIA Jetson Nano.

Intel's Movidius NCS is a USB thumb drive sized deep learning coprocessor. You inset the USB drive into a RPi and access it via the OpenVINO toolkit. The NCS can run between 80-150 GFLOPs with just over 1W of power, enabling embedded devices, such as the RPi, to run state-of-the-art neural networks.

Similar to the Movidius NCS, the Google Coral USB Accelerator is also a coprocessor device that can similarly be inserted in to a RPi via a USB port, and just like the NCS, it is designed only for inference (i.e. you wouldn't train a model with either the Coral or NCS). Google reports that their Coral line of products are over 10x faster than the NCS.

Just like there are times when you may need a coprocessor to obtain adequate throughput speeds for your deep learning and computer vision pipeline, there are also times where you may need to abandon the Raspberry Pi altogether and instead utilize a dedicated development board.

Currently, there are two frontrunners in the dedicated deep learning development board market: the Google Coral TPU Dev Board and the NVIDIA Jetson Nano. Unlike the base Coral USB Accelerator, the Coral TPU Dev Board is a dedicated board capable of up to 32-64 GFLOPs - far more powerful than both the TPU USB Accelerator and the Movidius NCS. The device itself utilises a highly optimised implementation of TensorFlow Lite, capable of running object detection models such as MobileNet V2 at 100+ FPS in a power efficient manner.

Competing with Google Coral's TPU Dev Board is the NVIDIA Jetson Nano. The Jetson Nano includes a 128-core Maxwell GPU, a quad-core ARM 57 processor running at 1.43 GHz, and 4GB of 64-bit LPDDR4 RAM. The Nano can provide 472 GFLOPs at only 5-10W of power.

6.4.2 Radio Frequency Spectrum Intelligence Gathering

AI methods continue to show enormous promise in improving Radio Frequency signal identification [43, 44]. The program manager of DARPA's Microsystems Technology Office, Paul Tilghman foresees the main aim of Radio Frequency AI systems to "see and understand the composition of the radio frequency spectrum - the kinds of signals occupying it, differentiating those that are 'important' from the background and identifying those that don't follow the rules" [45].

7 Conclusion

Based on the above discussion it can be concluded that applying AI techniques to cybersecurity is a challenging task particularly in areas such as Network Intrusion Detection, Malware Detection and Phishing Detection. The challenge is made more difficult from the lack of good quality public datasets that Cybersecurity AI researchers could use to develop suitable models. Machine Learning and by extension, Deep Learning methods cannot work without representative datasets, of which it is difficult and time consuming to obtain. However, cybersecurity vendors have invested resources to collect data sets for their particular products, but in most cases details of the datasets and models are unknown.

This paper has provided a brief introduction to deep layered models, autoencoder models, convolution models, NLP models and how to train these models, providing useful knowledge which can be used when assessing potential vendors' products. In addition to this, an insight into the use of open source image classification and object detection models in AI based security surveillance systems has also been provided, highlighting the greatly enhanced capabilities of the security surveillance domain, which now requires minimal human interaction.

8 References

[1] Rosenblatt, F. (1958). *The Perceptron: A Probabilistic Model for Information Storage and Organization.*

[2] Chollet, F. (2017). *Deep learning with python.*

[3] Nkiama, H. et al. (2016). *A Subset Feature Elimination Mechanism for Intrusion Detection System.*

[4] Dhanabal, L. et al. (2015). *A Study on NSL-KDD Dataset for Intrusion Detection System Based on Classification Algorithm.*

[5] Tavallaee, M. et al. (2009). *A Detailed Analysis of the KDD CUP 99 Data Set.*

[6] Revathi, S. et al. (2013). *A Detailed Analysis on NSL-KDD Dataset Using Various Machine Learning Techniques for Intrusion Detection.*

[7] Noureldien, N.A. et al. (2016). *Accuracy of Machine Learning Algorithms in Detecting DoS Attacks Types.*

[8] Stanford University, *CS231n: Convolutional Neural Networks for Visual Recognition, http://cs231n.stanford.edu.*

[9] Niyaz, Q. et al. (2015). *A Deep Learning Approach for Network Intrusion Detection System.*

[10] Shone, N. et al. (2017). *A Deep Learning Approach for Network Intrusion Detection.*

[11] Lopez-Martin, M. et al. (2017). *Conditional Variational Autoencoder for Prediction and Feature Recovery Applied to Intrusion Detection in IoT.*

[12] Yu, Y. et al. (2017). *Network Intrusion Detection through Stacking Dilated Convolutional Autoencoders.*

[13] Mirsky, Y. et al. (2018). *Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection.*

[14] Dong, B. et al. (2016). *Comparison Deep Learning Method to Traditional Methods Using for Network Intrusion Detection.*

[15] Yousefi-Azar, M. et al. (2017). *Autoencoder-based Feature Learning for Cyber Security Applications.*

[16] Ronen, R. et al. (2018). *Microsoft Malware Classification Challenge.*

[17] Ranveer, S. et al. (2015). *Comparative Analysis of Feature Extraction Methods of Malware Detection.*

[18] Zhang, J. (2019). *Machine Learning with Feature Selection Using Principal Component Analysis Malware Detection: A Case Study.*

[19] Nataraj, L. et al. (2011). *Malware Images: Visualization and Automatic Classification.*

[20] Gilbert, D. et al. (2018) *Using convolutional neural networks for classification of malware represented as images.*

[21] Kabanga, E. et al. (2018). *Malware Images Classification Using Convolutional Neural Network.*

[22] Parmuval, P. et al. (2018). *Malware Family Detection Approach using Image Processing Techniques: Visualization Technique.*

[23] Makandar, A. et al. (2015). *Overview of Malware Analysis and Detection.*

[24] Kaggle & Microsoft. (2015). *Microsoft Malware Classification Challenge (BIG 2015*), *https://www.kaggle.com/c/malware-classification/overview.*

[25] Stanford University, *Convolutional Neural Networks for Visual Recognition, https://cs231n.github.io/convolutional-networks/.*

[26] Chatterjee, M. et al. (2019). *Deep Reinforcement Learning for Detecting Malicious Websites.*

[27] Center for Machine Learning and Intelligent Systems, *SMS spam collection*, *https://archive.ics.uci.edu/ml/machine-learning-databases/00228/smsspamcollection.zip*.

[28] Froiland, J. (2018). *Recurrent Neural Networks: Part 2*, *https://medium.com/@jon.froiland/recurrent-neural-networks-part-2-723bc76ef36f*.

[29] Pennington, J. (2014). *GloVe: Global Vectors for Word Representation*, *https://nlp.stanford.edu/projects/glove/*.

[30] Brownlee, J. (2019). *A Gentle Introduction to Object Recognition with Deep Learning*, *https://machinelearningmastery.com/object-recognition-with-deep-learning/*.

[31] Rosebrock, A. (2018). *YOLO object detection with OpenCV*. *https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/*.

[32] Russakovsky, O. et al. (2015). *ImageNet Large Scale Visual Recognition Challenge*, *http://www.image-net.org/challenges/LSVRC/*.

[33] Howard, A.G. et al. (2017). *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*.

[34] Olmos, R. et al. (2017). *Automatic Handgun Detection Alarm in Videos Using Deep Learning*.

[35] FIZYR, *Keras RetinaNet*, *https://github.com/fizyr/keras-retinanet*.

[36] Tensorflow, *Tensorflow Object Detection API*, *https://github.com/tensorflow/models/tree/master/research/object_detection*.

[37] Ngoc Anh, H. *YOLO3 (Detection, Training, and Evaluation)*. *https://github.com/experiencor/keras-yolo3*.

[38] Wheeler, F. et al. (2011). *Handbook of Face Recognition*.

[39] Wang, M. et al. (2018). *Deep Face Recognition: A Survey*.

[40] Visual Geometry Group, Department of Engineering Science, University of Oxford. (2018). *VGGFace2 Dataset for Face Recognition, https://github.com/ox-vgg/vgg_face2*.

[41] Google Coral, *https://coral.ai/*.

[42] Intel Neural Compute Stick 2, *https://software.intel.com/en-us/neural-compute-stick*.

[43] Machine Learning and RF Spectrum Intelligence Gathering, *https://www.crfs.com/white-papers/machine-learning-rf-spectrum-white-paper*.

[44] DeepSig Inc, *https://www.deepsig.io/publications*.

[45] DARPA, (2017). *The Radio Frequency Spectrum + Machine Learning = A New Wave in Radio Technology, https://www.darpa.mil/news-events/2017-08-11a*.