A Selective Survey of Deep Learning Techniques and Their Application to Malware Analysis

Mark Stamp

Abstract In this chapter, we consider neural networks and deep learning, within the context of malware research. A variety of architectures are introduced, including multilayer perceptrons (MLP), convolutional neural networks (CNN), recurrent neural networks (RNN), long short-term memory (LSTM), residual networks (ResNet), generative adversarial networks (GAN), and Word2Vec. We provide a selective survey of applications of each of these architectures to malware-related problems.

1 Introduction

In this chapter, we discuss a variety of topics related to deep learning, with the primary focus on popular neural networking-based architectures. We survey various malware-related applications of each architecture considered. Each topic is discussed in some detail, with additional references for further reading provided in all cases.

This chapter can be viewed as a companion to the survey [78], which covers classic machine learning techniques and their applications in cybersecurity research. Our focus here is on neural networks and deep learning, and with respect to applications, we focus most of our attention on malware-related topics, but we do mention other applications within the broader information security domain.

For the sake of completeness, we begin with an introduction to artificial neural networks (ANNs), which includes a brief history of neural networks. We then introduce a wide variety of architectures and techniques, including convolutional neural networks (CNN), recurrent neural networks (RNN), long short-term memory (LSTM), residual networks (ResNet), and generative adversarial networks (GAN). We also discuss related techniques, such as word embeddings—including Word2Vec. We also briefly mention ensemble techniques and transfer learning in passing.

M. Stamp (✉)
San Jose State University, San Jose, CA, USA
e-mail: mark.stamp@sjsu.edu

M. Stamp et al. (eds.), *Malware Analysis Using Artificial Intelligence and Deep Learning*, https://doi.org/10.1007/978-3-030-62582-5_1

2 A Brief History of ANNs

The concept of an artificial neuron [26, 82] is not new, as the idea was proposed by McCulloch and Pitts in the 1940s [52]. However, modern computational neural networking really begins with the perceptron, which was first proposed by Rosenblatt in the late 1950s [68].

An artificial neuron with three inputs is illustrated in Fig. 1. In the original McCulloch-Pitts formulation, $X_i \in \{0, 1\}$, $w_i \in \{+1, -1\}$, and the output $Y \in \{0, 1\}$. The threshold T determines whether the output Y is 0 (inactive) or 1 (active), based on $\sum w_i X_i$. The thinking was that a neuron either fires or it does not (thus, $Y \in \{0, 1\}$), and the inputs would come from other neurons (thus, $X_i \in \{0, 1\}$), while the weights w_i specify whether an input is excitatory (increasing the chance of the neuron firing) or inhibitory (decreasing the chance of the neuron firing). Whenever $\sum w_i X_i > T$, the excitatory response wins, and the neuron fires; otherwise, the inhibitory response wins and the neuron does not fire.

A *perceptron* is considerably less restrictive than a McCulloch–Pitts artificial neuron, as the X_i and w_i can be real-valued. Since we want to use a perceptron as a binary classifier, the output is generally taken to be binary. McCulloch and Pitts chose such a restrictive formulation because they were trying to model logic functions. At the time, it was felt that encoding elementary logic into artificial neurons would be the key step to constructing systems with artificial intelligence. However, that point of view has certainly not panned out, while the additional generality offered by the perceptron formulation has proven extremely useful.

Given a real-valued input vector $X = (X_0, X_1, \ldots, X_{n-1})$, a perceptron can be viewed as a function of the form

$$f(X) = \sum_{i=0}^{n-1} w_i X_i + b,$$

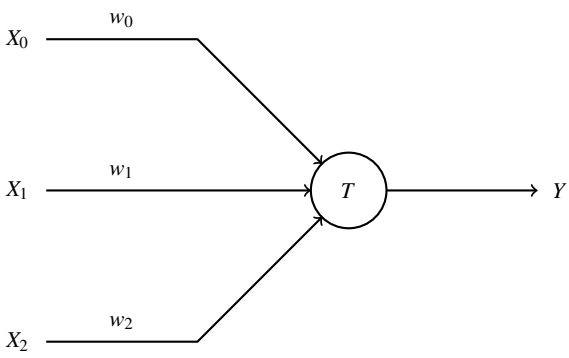


Fig. 1 Artificial neuron

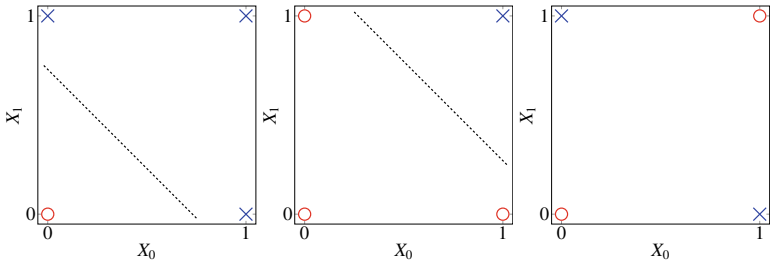


Fig. 2 OR and AND are linearly separable but XOR is not

that is, a perceptron computes a weighted sum of the components. Based on a threshold, a perceptron can be used to define a binary classifier. For example, we could classify a sample X as "type 1" provided that $f(X) > T$, for some specified threshold T, and otherwise classify X as "type 0."

In the case of two-dimensional input, the decision boundary of a perceptron defines a line

$$f(x, y) = w_0 x + w_1 y + b. \tag{1}$$

It follows that a perceptron cannot provide ideal separation in cases where the data itself is not linearly separable.

There was considerable research into ANNs in the 1950s and 1960s, and that era is often described as the first "golden age" of AI and neural networks. But the gold turned to lead in 1969 when an influential work by Minsky and Papert [55] emphasized the limitations of perceptrons. Specifically, they observed that the XOR function is not linearly separable, which implies that a single perceptron cannot model something as elementary as XOR. The OR, AND, and XOR functions are illustrated in Fig. 2, where we see that OR and AND are linearly separable, while XOR is not.

As the name suggests, a multilayer perceptron (MLP) is an ANN that includes multiple (hidden) layers in the form of perceptrons. An example of an MLP with two hidden layers is given in Fig. 3, where each edge represents a weight that is to be determined. Unlike a single-layer perceptron, MLPs are not restricted to linear decision boundaries, and hence an MLP can accurately model the XOR function. However, the perceptron training method proposed by Rosenblatt [68] cannot be used to effectively train an MLP [44]. To train a single perceptron, simple heuristics will suffice, assuming that the data is linearly separable. From a high-level perspective, training a single perceptron is somewhat analogous to training a linear SVM, except that for a perceptron, we do not require that the margin (i.e., minimum separation) be maximized. However, training an MLP would appear to be challenging since we have hidden layers between the input and output, and it is not clear how changes to the weights in these hidden layers will affect each other, let alone the output.

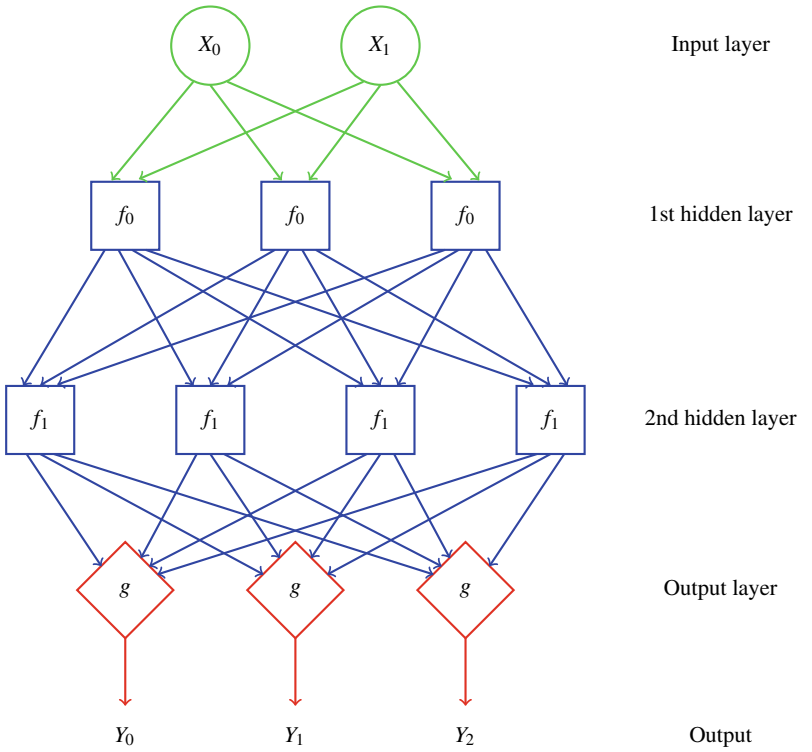


Fig. 3 MLP with two hidden layers

As an aside, it is interesting to note that for SVMs, we deal with data that is not linearly separable by employing a soft margin (i.e., we allow for training errors) and by the use of the so-called "kernel trick," where we map the input data to a higher dimensional feature space using a (nonlinear) kernel function. In contrast, perceptrons (in the form of MLPs) overcome the limitation of linear separability by the use of multiple layers. For an MLP, it is almost as if the nonlinear kernel function has been embedded directly into the model itself through the use of hidden layers, as opposed to a user-specified explicit kernel function, as is the case for an SVM.

One possible advantage of the MLP approach over an SVM is that for an MLP, the equivalent of the kernel function is, in effect, derived from the data and refined through the training process. In contrast, for an SVM, the kernel function is selected by a human, and once selected it does not change. In machine learning, removing those pesky humans from the learning process is a good thing. However, a possible tradeoff is that significantly more training data will likely be needed for an MLP, as compared to an SVM, due to the greater data requirement involved in learning the equivalent of a kernel function.

As another aside, we note that from a high-level perspective, it is possible to view MLPs as combining some aspects of SVMs (i.e., specifically, nonlinear decision boundaries) and HMMs (i.e., hidden layers). Also, we will see that the backpropagation algorithm that is used to train MLPs includes a forward pass and backward pass, which is eerily reminiscent of the training process that is used for HMMs.

As yet another aside, we note that an MLP is a *feedforward neural network*, which means that there are no loops—the input data and intermediate results feed directly through the network. In contrast, a recurrent neural network (RNN) can have loops, which gives an RNN a concept of memory but can also add significant complexity.

In the book *Perceptrons: An Introduction to Computational Geometry*, published in 1969, Minsky and Papert [55] made much of the perceived shortcoming of perceptrons—in particular, the aforementioned inability to model XOR. This was widely viewed as a devastating criticism at the time, as it was believed that successful AI would need to capture basic principles of logic. Although it was known that perceptrons with multiple layers (i.e., MLPs) can model XOR, at the time, nobody knew how to efficiently train MLPs. Minsky and Papert's work was highly influential and is frequently blamed for the relative lack of interest in the field—a so-called "AI winter"—that persisted throughout the 1970s and into the early 1980s.

By 1986, there was renewed interest in ANNs, thanks in large part to the work of Rumelhart, Hinton, and Williams [70], who developed a practical means of training MLPs—the method of backpropagation. For details on backpropagation, see [80], for example.

It is worth noting that there was another "AI winter" that lasted from the late 1980s through the early 1990s (at least). The proximate cause of this most recent AI winter was that the hype far outran the limited successes that had been achieved. Although deep learning has now brought ANNs back into vogue, your author (a doubting Thomas, and proud of it) is not convinced that the current artificial intelligence mania will prove any less artificial than previous AI "summers" which, on the whole, yielded mostly disappointment. Some of the ridiculous statements being made today [28] lead your eminently sensible author to believe that the hype is already hopelessly out of control.[1]

Next, we discuss deep learning, which builds on the foundation of ANNs. We can view the relationship between ANNs and deep learning as being somewhat akin to that of Markov chains and HMMs, for example. That is, ANNs serve as a basic technology that can be used to build a powerful machine learning technique, analogous to the way that an HMM is built on the foundation of an elementary Markov chain. But, before we get into the details of deep learning, we consider the topic from a high-level perspective.

---

[1] In stark contrast to the nonsensical hype that envelopes far too much of the discussion of deep learning and (especially) AI, there does exist some clear-headed thinking that points to the great transformative potential of learning technology in the real world, rather than the world of science fiction. For a fine example of this latter genre, see the intriguingly titled article, "Models will run the world" [14]. (Spoiler alert: "Models will run the world" is *not* about world domination by skinny women in swimsuits).

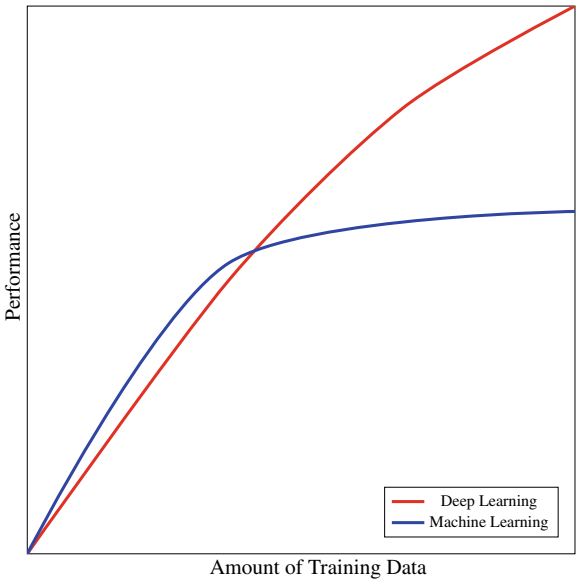


Fig. 4 Model performance as a function of the amount of training data

3 Why Deep Learning?

It is sometimes claimed that the major advantage of deep learning arises when the amount of training data is large. For example, the tutorial [35] gives a graph similar to that in Fig. 4, which purports to show that deep learning will continue to achieve improved results as the size of the dataset grows, whereas other machine learning techniques will plateau at some relatively early point. That is, models generated by non-deep learning techniques will "saturate" relatively quickly, and once this saturation point is reached, more data will not yield improved models.[2] In contrast, deep learning is supposed to continue learning, essentially without limit as the volume of training data increases, or at least it will plateau at a much higher level. Of course, even if this is entirely true, there are practical computational constraints since more data requires more computing power for training.

[2]If any learning model truly saturates, then adding more data will be counterproductive beyond some point, as the work factor for training on larger datasets increases, while there is no added benefit from the resulting trained model. It would therefore be useful to be able to predetermine a "score" of some sort that would tell us approximately how much data is optimal when training a particular learning model for a given type of data.

4 Decisions, Decisions

The essence of machine learning is that when training a model, we minimize the need for input from those fallible humans. That is, we want our machine learning models to be *data-driven*, in the sense that the models learn as much as possible directly from the data itself, with minimal human intervention. However, any machine learning technique will require some human decisions—for HMMs, we specify the number of hidden states; for SVMs, we specify the kernel function; and so on.

For ANNs in general, and deep learning in particular, the following design decisions are relevant [22].

- The *depth* of an ANN refers to the number of hidden layers. The "deep" in deep learning indicates that we employ ANNs with lots of hidden layers, where "lots" seems to generally mean as many as possible, based on available computing power.
- The *width* of an ANN is the number of neurons per layer, which need not be the same in each layer.
- In an MLP, for example, nonlinearity is necessary, and this is achieved through the *activation functions* (also known as transfer functions). Most activation functions used in deep learning are designed to mimic a step function—examples include the sigmoid (or logistic) function

$$f(x) = \frac{1}{1 + e^{-x}},$$

the hyperbolic tangent

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}},$$

the inverse tangent (also known as arctangent)

$$f(x) = \tan^{-1}(x),$$

and the rectified linear unit (ReLU)

$$f(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Note that the softmax function is a generalization of the sigmoid function to multiclass problems.

The graph of each of the activation functions given above is illustrated in Fig. 5. As of this writing, ReLU is the most popular activation function. Numerous variants of the ReLU function are also used, including the leaky ReLU and exponential linear unit (ELU).

- In addition to activation functions, we also specify an *objective function*. The objective function is the function that we are trying to optimize and typically represents the training error.
- A *bias node* may be included (or not) in any hidden layer. Each bias node generates a constant value and hence is not connected to any previous layer. When present, a bias node allows the activation function to be shifted. In the perceptron example given in (1), the bias corresponds to the y-intercept b.

For the sake of comparison with our favorite non-deep learning technique, the depth of an HMM can be viewed as the order of the underlying Markov model. Typically, for HMMs, we only consider models of order one (in which case, the current state depends only on the previous state), but it is possible to consider higher order models. The width of an HMM might be viewed as being determined by N, the number of hidden states. But, regardless of the order of the model or the choice of N, there is really only one hidden layer in any HMM. The fact that an HMM is based on linear operations implies that adding multiple hidden layers would have no effect, as the multiple layers would be equivalent to a single layer. Furthermore, the A and B matrices of an HMM can be viewed as its activation functions (with the B matrix corresponding to the output layer), and $P(\mathcal{O} \mid \lambda)$ corresponds to the objective function in an ANN. Note that these functions are all linear in an HMM, while at least some of the activation functions must be nonlinear in any true multilayer ANN, such as an MLP.

Neural networks are trained using the backpropagation algorithm, which is a special case of a more general technique known as reverse mode automatic differentiation. For additional details on the topic of backpropagation, see, for example, [80].

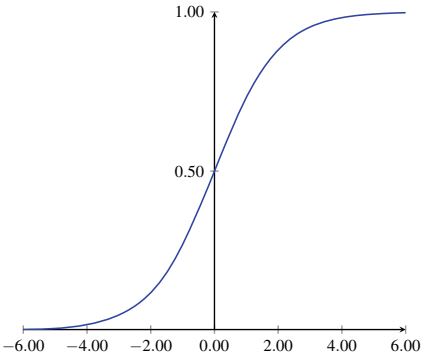
The remainder of this paper is focused on various neural network based architectures and related topics. For each topic covered, we discuss research in the field of malware analysis.

5 Multilayer Perceptrons

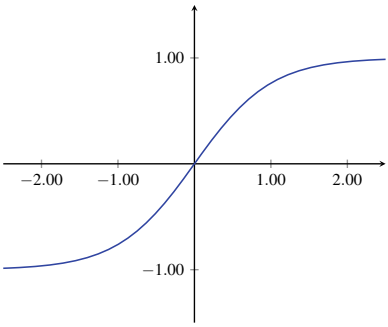
We have already discussed multilayer perceptrons (MLP) in some detail. MLPs are in some sense one of the most generic neural networking architectures—when someone speaks of a neural network in general, there is a good chance that they have an MLP in mind.

5.1 Overview of MLPs

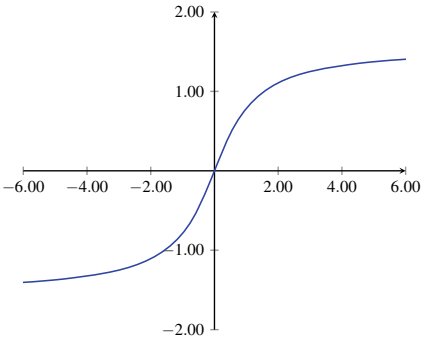
Recall that Fig. 3 is an example of an MLP with two hidden layers. Each edge in the figure represents a weight that is to be determined via training, and backpropagation



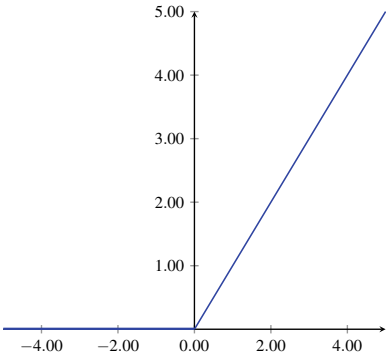
(a) Sigmoid function



(b) Hyperbolic tangent



(c) Arctangent



(d) ReLU

Fig. 5 Activation functions

is an efficient and effective way to train such a network. The advantage of an MLP is that it is not restricted to linear decision boundary.

5.2 MLPs in Malware Analysis

MLPs are extremely popular, and in most fields, they are one of the first learning techniques considered. Information security is no exception, as MLPs have been applied to nearly every security problem where deep learning techniques are applicable. Not surprisingly, large numbers of malware research papers employ MLPs. For example, in [5] MLPs are trained on progressively more generic malware families, yielding quantifiable results on the inherent tradeoff between the generality of the training data and accuracy. The research in [74] shows that a straightforward ensemble of various learning algorithms—including MLPs—can generate significantly stronger

results than any of the component techniques. The paper [85] uses MLPs as part of an Android malware detection technique.

Another field in information security where MLPs have played a very prominent role is in intrusion detection systems (IDS). For example, the paper [57] uses MLPs in a novel multiclass IDS approach.

6 Convolutional Neural Networks

In this section, we provide an introduction to one of the most important and widely used learning techniques—CNN. After a brief overview, we introduce discrete convolutions with the focus on their specific application to CNNs. We then consider a simplified example that serves to illustrate various aspects of CNNs.

6.1 Overview of CNNs

Generically, ANNs use fully connected layers. A fully connected layer can deal effectively with correlations between any points within the training vectors, regardless of whether those points are close together, far apart, or somewhere in between. In contrast, a CNN, is designed to deal with local structure—a convolutional layer cannot be expected to perform well when crucial information is not local. A key benefit of CNNs is that convolutional layers can be trained much more efficiently than fully connected layers.

For images, most of the important structure (edges and gradients, for example) is local. Hence, CNNs would seem to be an ideal tool for image analysis and, in fact, CNNs were developed for precisely this problem. However, CNNs have performed well in a variety of other problem domains. In general, any problem for which there exists a data representation where local structure predominates is a candidate for a CNN. In addition to images, local structure is crucial in fields such as text analysis and speech analysis, for example.

6.2 Convolutions and CNNs

A discrete convolution is a sequence that is itself a composition of two sequences and is computed as a sum of pointwise products. Let $c = x * y$ denote the convolution of sequences $x = (x_0, x_1.x_2, \ldots)$ and $y = (y_0, y_1.y_2, \ldots)$. Then the k^{th} element of the convolution is given by

$$c_k = \sum_{k=i+j} x_i y_j = \sum_i x_i y_{k-i}.$$

We can view this process as x being a "filter" (or kernel) that is applied to the sequence y over a sliding window.

For example, if $x = (x_0, x_1)$ and $y = (y_0, y_1, y_2, y_3, y_4)$, we find

$$c = x * y = \big(x_0y_1 + x_1y_0, x_0y_2 + x_1y_1, x_0y_3 + x_1y_2, x_0y_4 + x_1y_3\big).$$

If we reverse the order of the elements of x, then we have

$$c = \big(x_0y_0 + x_1y_1, x_0y_1 + x_1y_2, x_0y_2 + x_1y_3, x_0y_3 + x_1y_4\big)$$

which is, perhaps, a slightly more natural and intuitive way to view the convolution operation.

Again, we can view x as a filter that is applied to the sequence y. Henceforth, we define this filtering operation as convolution with the order of the elements of the filter reversed. For example, suppose that we apply the filter $x = (1, -2)$ to the sequence $y = (0, 1, 2, 3, 4)$. In this case, the convolution gives us

$$\begin{aligned} c = x * y &= \big(x_0y_0 + x_1y_1, x_0y_1 + x_1y_2, x_0y_2 + x_1y_3, x_0y_3 + x_1y_4\big) \\ &= \big(1 \cdot 0 - 2 \cdot 1, 1 \cdot 1 - 2 \cdot 2, 1 \cdot 2 - 2 \cdot 3, 1 \cdot 3 - 2 \cdot 4\big) \\ &= \big(-2, -3, -4, -5\big) \end{aligned}$$

We can define an analogous filtering (or discrete convolution) operation in two or three dimensions. For the two-dimensional case, suppose that $A = \{a_{ij}\}$ is an $N \times M$ matrix representing an image and $F = \{f_{ij}\}$ is an $n \times m$ filter. Let $C = \{c_{ij}\}$ be the convolution of F with A. As in the one-dimensional case, we denote this convolution as $C = F * A$. In this two-dimensional case, we have

$$c_{ij} = \sum_{k=0}^{n-1} \sum_{\ell=0}^{m-1} f_{k,\ell} a_{i+k,j+\ell},$$

where $i = 0, 1, \ldots, N - n$ and $j = 0, 1, \ldots, M - m$. That is, we simply apply the filter F at each offset of A to create the new—and slightly smaller—matrix that we denote as C. The three-dimensional case is completely analogous to the two-dimensional case.

We could simply define filters as we see fit, with each filter designed to correspond to a specific feature.[3] But since we are machine learning aficionados, for CNNs, we let the data itself determine the filters. Therefore, training a CNN can be viewed as determining filters, based on the training data. As with any respectable neural network, we can train CNNs via backpropagation.

Suppose that A represents an image and we train a CNN on the image A. Then the first convolutional layer is trained directly on the image. The filters determined at this first layer will correspond to fairly intuitive features, such as edges, basic

[3] We see examples of filters applied to simple images in Sect. 6.3.

shapes, and so on. We can then apply a second convolutional layer, that is, we apply a similar convolutional process, but the output of the first convolutional layer is the input to this second layer. At the second layer, filters are trained based on features of features. Perhaps not surprisingly, these second layer filters correspond to more abstract features of the original image A, such as the "texture." We can repeat this convolution of convolutions step again and again, at each layer obtaining filters that correspond to features representing a higher degree of abstraction, as compared to the previous layer. The final layer of a CNN is not a convolution layer but is instead a typical fully-connected layer that can be used to classify based on complex image characteristics (e.g., "cat" versus "dog"). In addition, so-called pooling layers can be used between some of the convolutional layers. Pooling layers are simple—no training is involved—and serve primarily to reduce the dimensionality of the problem. Below, we give a simple example that includes a pooling layer.

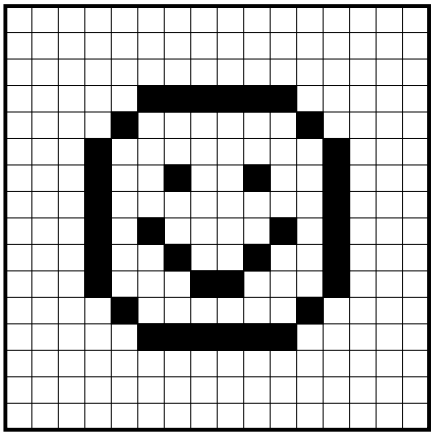
In addition to having multiple convolutional layers, at each layer, we can (and generally will) stack several convolutions on top of each other. These filters are all initialized randomly, so they can all learn different features. In fact, for a typical color image, the image itself can be viewed as consisting of three layers, corresponding to the R, G, and B components in the RGB color scheme. Hence, for color images, the filters for the first convolutional layer will be three-dimensional, while subsequent convolutional layers can—and, typically, will—be three-dimensional as well, due to the stacking of multiple convolutions/filters at each layer. For simplicity, in our example, we only consider a black-and-white two-dimensional image, and we only apply one convolution at each layer.

Before considering a simple example, we note that there are advantages of CNNs that are particularly relevant in the case of image analysis. For a generic neural network, each pixel would typically be treated as a separate neuron, and for any reasonable size of image, this would result in a huge number of parameters, making training impractical. In contrast, at the first layer of a CNN, each filter is applied over the entire image, and at subsequent layers, we apply filters over the entire output of the previous layer. One effect of this approach is that it greatly reduces the number of parameters that need to be learned. Furthermore, by sliding the filter across the image as a convolution, we obtain a degree of translation invariance, i.e., we can detect image features that appears at different offsets. This can be viewed as reducing the overfitting that would otherwise likely occur.

The bottom line is that CNNs represent an efficient and effective technique that was developed specifically for image analysis. However, CNNs are not restricted to image data, and can be useful in any problem domain where local structure is dominant.

6.3 Example CNN

Now we turn our attention to a simple example that serves to illustrate some of the points discussed above. Suppose that we are dealing with black-and-white images,



(a) Pixels

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Numeric pixel values

Fig. 6 A 16 × 16 black-and-white image

2	−1	−1
−1	2	−1
−1	−1	2

(a) Diagonal

−1	2	−1
−1	2	−1
−1	2	−1

(b) Vertical

−1	−1	−1
2	2	2
−1	−1	−1

(c) Horizontal

−1	−1	2
−1	2	−1
2	−1	−1

(d) Anti-diagonal

Fig. 7 Examples of filters

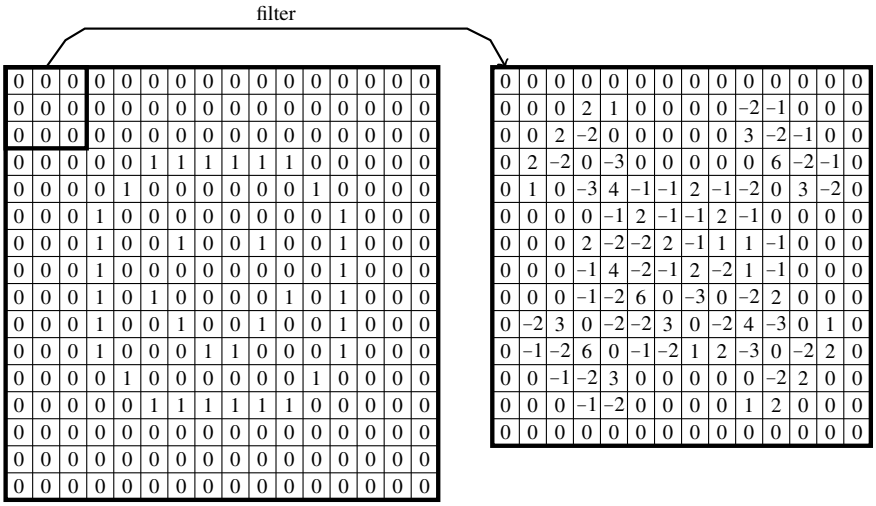
where each pixel is either 0 or 1, with 0 representing white and 1 representing black.[4] Further, suppose that the black-and-white images under consideration are 16 × 16 pixels in size. An example of such an image appears in Fig. 6.

In Fig. 7, we give some 3 × 3 filters. For example, the output of the filter in Fig. 7a is maximized when it aligns with a diagonal segment. Figure 8 shows the result of applying the convolution represented by the filter in Fig. 7a to the smiley face image in Fig. 6.

We note that, for the convolution in Fig. 8, the maximum value of 6 does indeed occur only at the three offsets where the (main) diagonal segments are all black and the off-diagonal elements are all white. These maximum values correspond to convolutions over the red boxes in Fig. 9.

In a CNN, so-called pooling layers are often intermixed with convolutional layers. As with a convolutional layer, in a pooling layer, we slide a window of a fixed size over the image. But whereas the filter in a convolutional layer is learned, in a pooling

[4]Color and grayscale images are more complex. For grayscale, a nonlinear encoding (i.e., gamma encoding) is employed, so as to make better use of the range of values available. For color images, the RGB (red, green, and blue, respectively) color scheme implies that each pixel is represented by 24 bits (in an uncompressed format), in which case convolutional filters can be viewed as operating over a three-dimensional box that is 3 bytes deep.

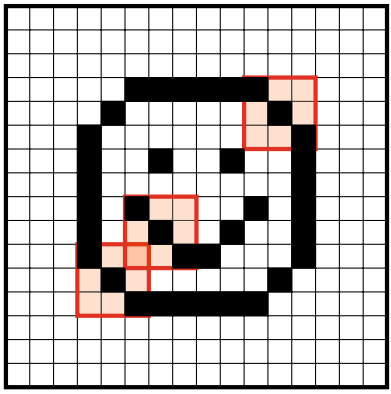


0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0
0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	0	1	1	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	2	1	0	0	0	0	−2	−1	0	0	0
0	0	2	−2	0	0	0	0	0	3	−2	−1	0	0
0	2	−2	0	−3	0	0	0	0	0	6	−2	−1	0
0	1	0	−3	4	−1	−1	2	−1	−2	0	3	−2	0
0	0	0	0	−1	2	−1	−1	2	−1	0	0	0	0
0	0	0	2	−2	−2	2	−1	1	1	−1	0	0	0
0	0	0	−1	4	−2	−1	2	−2	1	−1	0	0	0
0	0	0	−1	−2	6	0	−3	0	−2	2	0	0	0
0	−2	3	0	−2	−2	3	0	−2	4	−3	0	1	0
0	−1	−2	6	0	−1	−2	1	2	−3	0	−2	2	0
0	0	−1	−2	3	0	0	0	0	0	−2	2	0	0
0	0	0	−1	−2	0	0	0	0	1	2	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 8 First convolutional layer (3×3 filter from Fig. 7a)

Fig. 9 Maximum convolution values in Fig. 8



layer an extremely simple filter is specified and remains unchanged throughout the training. As the name implies, in *max pooling*, we simply take the the maximum value within the filter window. An illustration of max pooling is given in Fig. 10.

Instead of a max pooling scheme, sometimes *average pooling* is used. In any case, pooling can be viewed as a downsampling operation, which has the effect of reducing the dimensionality, and thus easing the computational burden of training subsequent convolutional layers.[5] To increase the downsampling effect, pooling usually uses non-overlapping windows. Note that the dimensionality reduction of pooling could

[5] It is also sometimes claimed that pooling improves certain desirable characteristics of CNNs, such as translation invariance and deformation stability. However, this is disputed, and the current trend seems to clearly be in the direction of fully convolutional architectures, i.e., CNNs with no pooling layers [69].

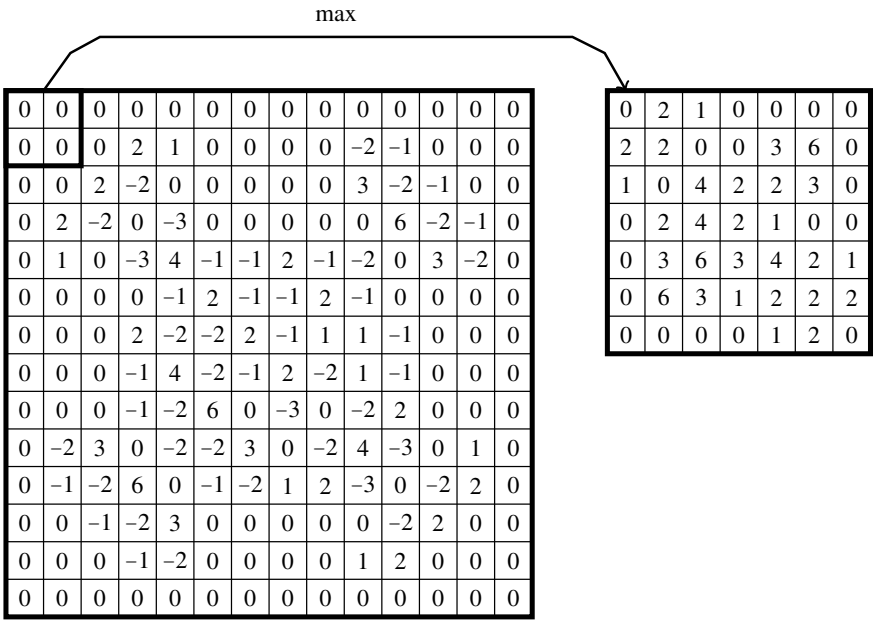


Fig. 10 Max pooling layer (2 × 2, non-overlapping)

also be achieved by a convolutional layer that uses a larger stride through the data, and in [75], for example, it is claimed that such an approach results in no loss in accuracy for the resulting CNN.

An illustration of the first convolutional layer for a color image is given in Fig. 11. In this case, a three-dimensional filter is applied over the R, G, and B components in the RGB color scheme. The example in Fig. 11 is meant to indicate that five different filters are being trained. Since each filter is initialized randomly, they can all learn different features. At the second convolutional layer, we can again train three-dimensional filters, based on the output of the first convolutional layer. This process is repeated for any additional convolutional layers.

There are several possible ways to visualize the filters in convolutional layers. For example, in [89], a de-convolution technique is used to obtain the results in Fig. 12. Here, each row is a randomly selected filter and the columns, from left to right, correspond to training epochs 1, 2, 5, 10, 20, 30, 40, and 64. From layer 4, we see that the training images must be faces. In general, it is apparent that the filters are learning progressively more abstract features as the layer increases.

A fairly detailed discussion of CNNs can be found at [38], while the paper [15] provides some interesting insights. For a more intuitive discussion, see [37], and if you want to see lots of nice pictures, take a look at [16]. More details on convolutions can be found in [61].

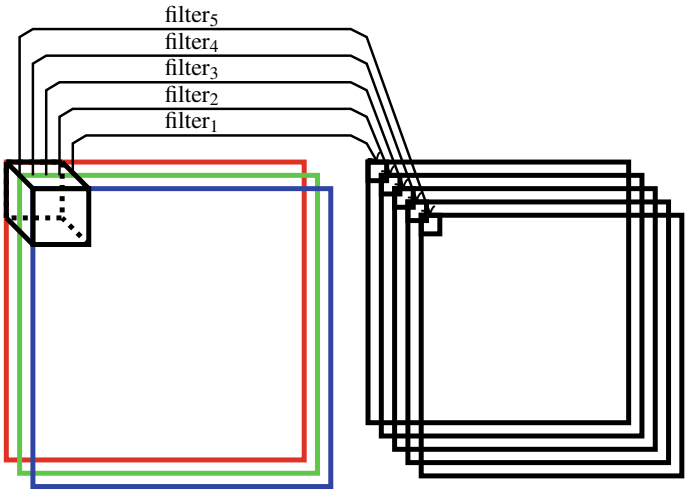


Fig. 11 First convolutional layer with stack of five filters (RGB image)

6.4 CNNs in Malware Analysis

CNNs have proven their worth in a wide variety of security-related applications. Some of these applications, such as image spam detection [1, 10, 72], are obvious and relatively straightforward applications of CNNs. However, other security domains that do not have any apparent image-based component have also had success with CNNs.

By treating executable files as images, researchers have been able to leverage the strengths of CNNs for malware detection, classification, and analysis. For example, the papers [6] and [88] treat executable files as images, and obtain the state-of-the-art result for the malware detection problem. In particular, the research in [88] makes extensive use of transfer learning, whereby the output layer of previously trained CNNs are retrained for the malware detection problem. This results in fast training times and very high malware classification accuracies.

The research in [34] compares CNNs to so-called extreme learning machines (ELM), a topic that we discuss below, in Sect. 10. The best CNN results in [34] are obtained using a one-dimensional CNN trained on the raw bytes of executable files. In [86], CNNs are successfully applied to a combination of static and dynamic features.

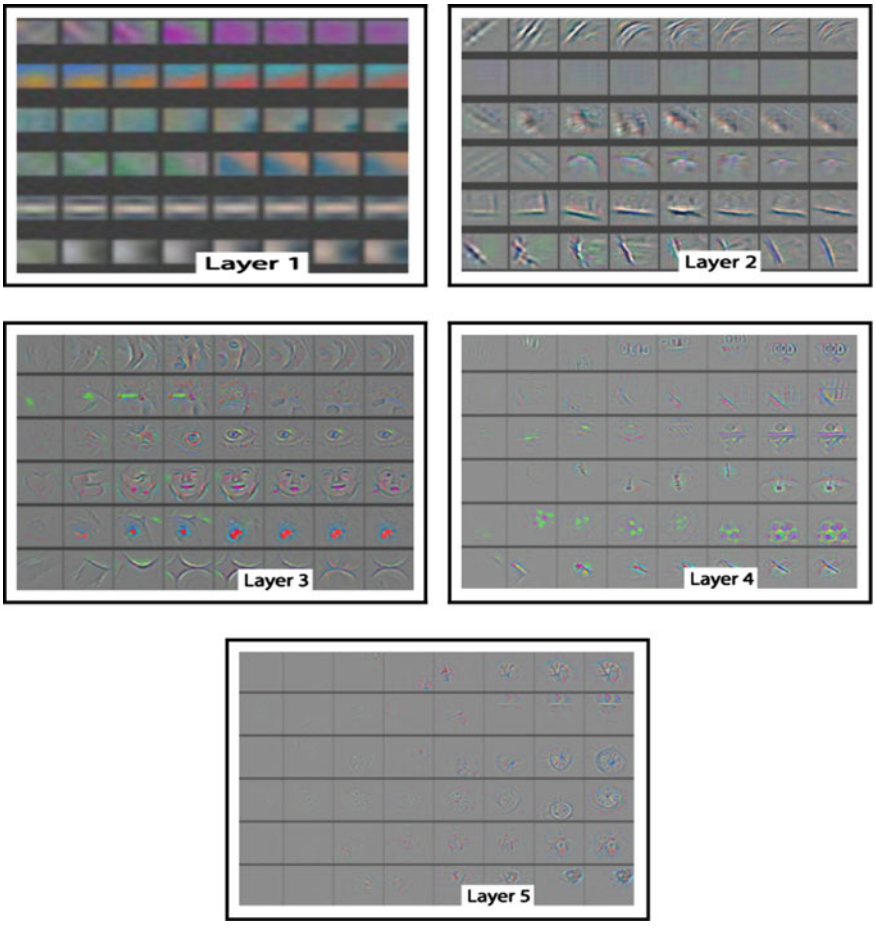


Fig. 12 Visualizing convolutions [89]

7 Recurrent Neural Networks

An example of a feedforward neural network with two hidden layers is given in Fig. 13. This type of neural network has no "memory" in the sense that each input vector is treated independently of other input vectors. Hence, such a feedforward network is not well suited to deal with sequential data.

In some cases, it is necessary for a classifier to have memory. For example, if we want to tag parts of speech in English text (i.e., noun–verb, and so on), this is not feasible if we only look at words in isolation. For example, the word "all" can be an adjective, adverb, noun, or even a pronoun, and the only way to determine which is the case is to consider the context. A recurrent neural network (RNN) provides a way to add memory (or context) to a feedforward neural network.

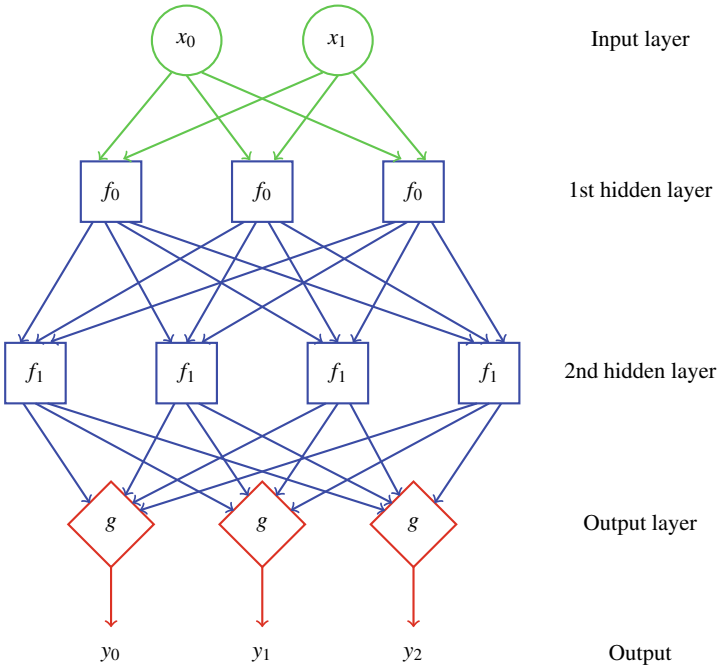


Fig. 13 Feedforward neural network with two hidden layers

To convert a feedforward neural network into an RNN, we treat the output of the hidden states as another input. For the neural network in Fig. 13, the corresponding generic RNN is illustrated in Fig. 14. The structure in Fig. 14 implies that there is a time step involved, that is, we train (and score) based on a sequence of input vectors. Of course, we cannot consider infinite sequences, and even if we could, the influence of feature vectors that occurred far back in time is likely to be minimal.

The RNN in Fig. 14 can be "unrolled," as illustrated in Fig. 15. Note that in this case, we use f to represent the hidden layer or layers, while the notation X_t is used to represent (x_0, x_1) at time step t from un-unrolled RNN in Fig. 14 and, similarly, Y_t corresponds to (y_0, y_1, y_2) at time t. From the unrolled form, it is clear that any RNN can be treated as a special case of a feedforward neural network, where the intermediate hidden layers (f in our notation) all have identical structures and weights. We can take advantage of this special structure to efficiently train an RNN using a (slight) variant of backpropagation, known as backpropagation through time (BPTT).

Before briefly turning our attention to BPTT, we illustrate some variants of a generic RNN. An RNN such as that illustrated in Fig. 15 is known as a sequence-to-sequence model, since each input sequence $(X_0, X_1, \ldots, X_{n-1})$ corresponds to an output sequence $(Y_0, Y_1, \ldots, Y_{n-1})$. In Fig. 16a, we have illustrated a many-to-one example of an RNN, that is, the case where an input sequence of the

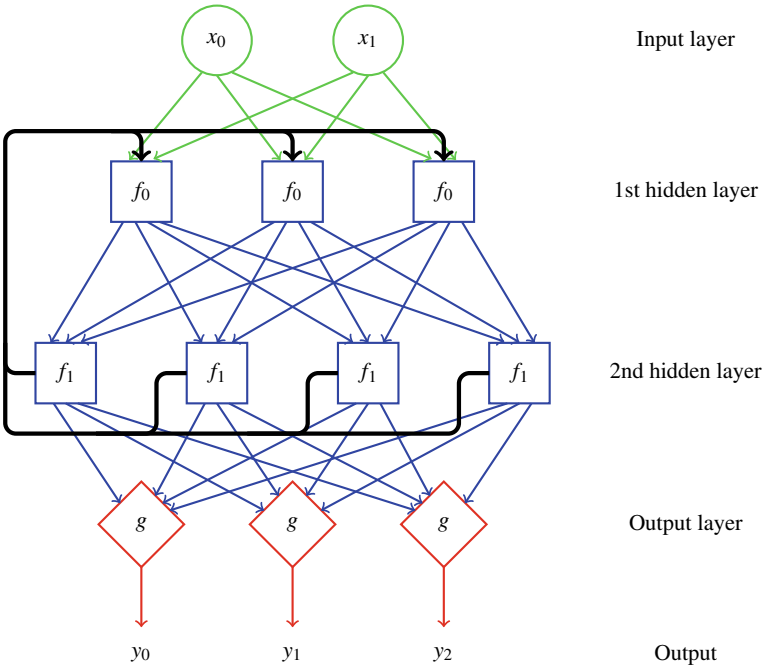


Fig. 14 Network in Fig. 13 as an RNN

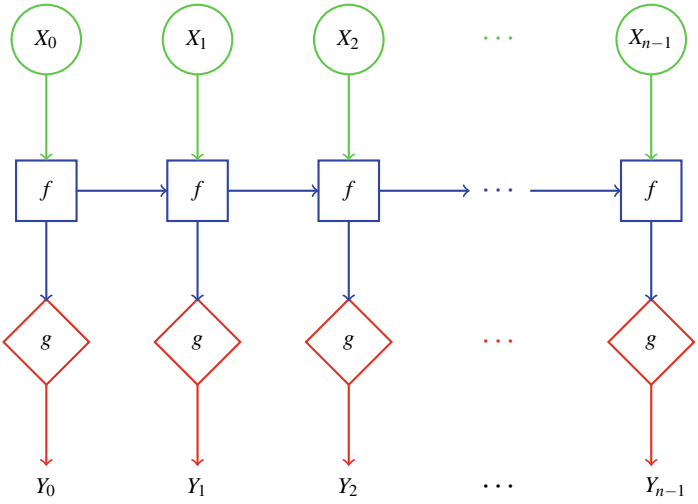
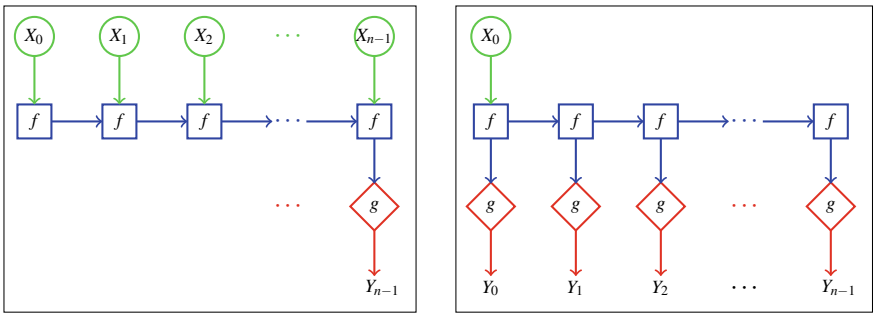


Fig. 15 Unrolled RNN (sequence-to-sequence model)



(a) Many-to-one RNN (b) One-to-many RNN

Fig. 16 Variants of the generic RNN in Fig. 15

form $(X_0, X_1, \ldots, X_{n-1})$ corresponds to the single output Y_{n-1}. At the other extreme, Fig. 16b illustrates a one-to-many RNN, where the single input X_0 corresponds to the output sequence $(Y_0, Y_1, \ldots, Y_{n-1})$.

A many-to-one model might be appropriate for part-of-speech tagging, for example, while a one-to-many RNN could be used for music generation. An example of an application where a sequence-to-sequence (or many-to-many) RNN would be appropriate is a machine translation. There are numerous possible variants of the sequence-to-sequence RNN. Also, note that a feedforward neural network, such as that in Fig. 13, can be viewed as a one-to-one RNN.

Multilayer RNNs can also be considered. This can be viewed as training multiple RNNs simultaneously, with the first RNN trained on the input data, the second RNN trained on the hidden states of the first RNN, and so on. A two-layer (sequence-to-sequence) RNN is illustrated in Fig. 17. Of course, more layers are possible, but the training complexity will increase, and hence only "shallow" RNN architectures (in terms of the number of layers) are generally considered.

7.1 Backpropagation Through Time

RNNs can be viewed as neural networks that are designed specifically for time series or other sequential data. With an RNN, the number of parameters is reduced so as to ease the training burden. This situation is somewhat analogous to CNNs, which are designed to efficiently deal with local structure (e.g., in images). That is, both CNNs and RNNs serve to make training more efficient—as compared to generic feedforward neural networks—for specific classes of problems. Backpropagation through time (BPTT) is simply an ever-so-slight variation on backpropagation that is optimized for training RNNs.

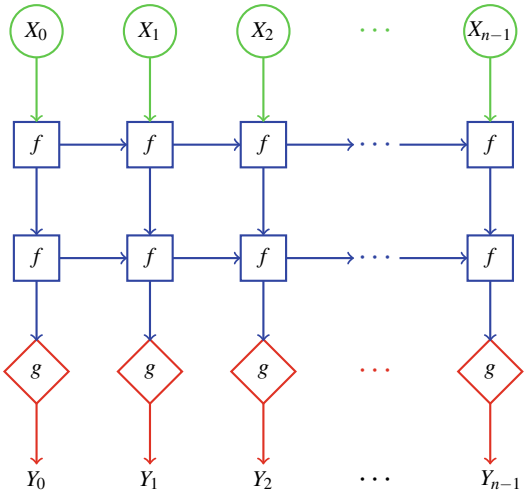


Fig. 17 Two-layer RNN

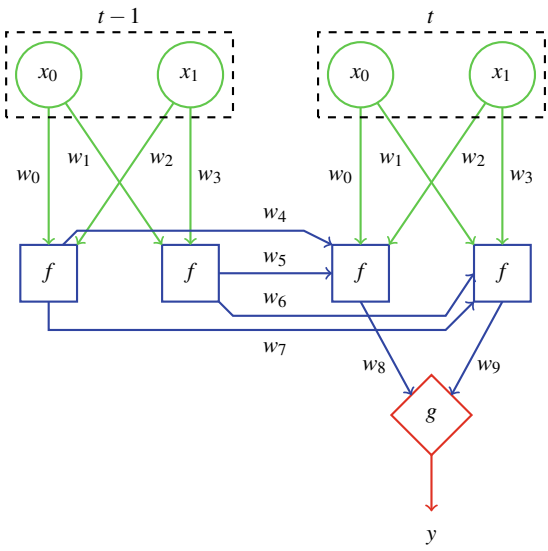


Fig. 18 Simple RNN example

In Fig. 18, we give a detailed view of a many-to-one (actually, two-to-one) RNN. In this case, we see that the 10 weights, $(w_0, w_1, \ldots, w_9)$ must be determined via training.

In Fig. 19, we give a neural network that is essentially the fully connected version of the RNN in Fig. 18. Note that in this fully connected version, there are 20 parameters to be determined. In an RNN, we assume that the data represents sequential input

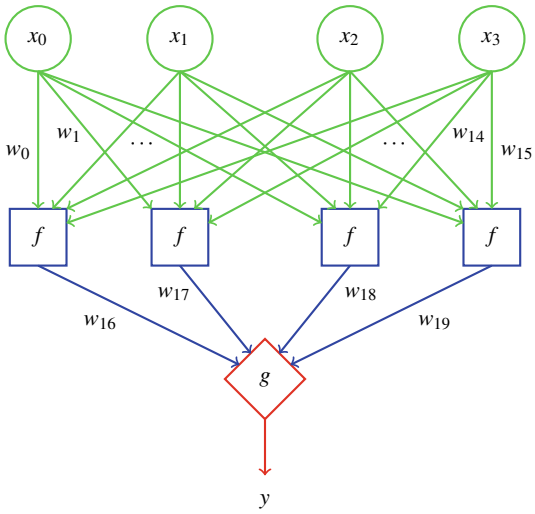


Fig. 19 Fully connected analog of Fig. 18

and hence the reduction in the number of weights is justified, since we are simply eliminating from consideration cases where the past is influenced by the future.[6]

It is well known that gradient issues are a concern when training neural networks in general, and are a particularly acute issue with generic RNNs. In an RNN, the further that we attempt to backpropagate through time, the more likely that the gradient will "explode" or "vanish" or oscillate between extremes. The details of the exploding gradient and vanishing gradient are beyond the scope of this survey; for more information on these topics, see [79], for example.

Next, we turn our attention to specialized RNN architectures that are designed to mitigate the gradient issues that plague generic RNNs. Specifically, we consider LSTM networks in some detail and we then briefly discuss a variant of LSTM known as gated recurrent units (GRU). In fact, a vast number of variants of the LSTM architecture have been developed. However, according to the extensive empirical study in [23], "none of the variants can improve upon the standard LSTM architecture significantly."

7.2 Long Short-Term Memory

In addition to being a tongue twister, LSTM networks are a class of RNN architectures that are designed to deal with long-range dependencies. That is, LSTM can deal with "gaps" between the appearance of a feature and the point at which it is needed by the model [23]. The claim to fame of LSTM is that it can reduce the effect of a

[6]Obviously, the inventors of RNNs were not familiar with *Back to the Future* or *Star Trek*, both of which conclusively demonstrate that the future can have a large influence on the past.

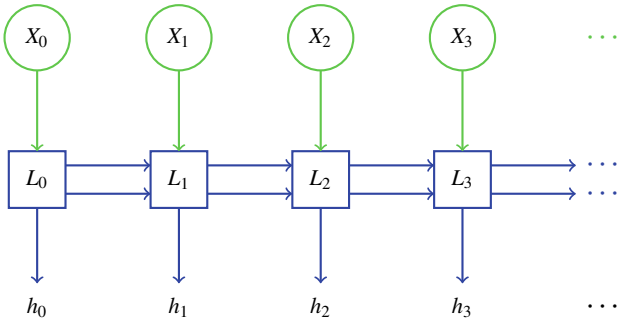


Fig. 20 LSTM

vanishing gradient, which is what enables such models to account for longer range dependencies [30].

Before outlining the main ideas behind LSTM, we note that the LSTM architecture has been one of the most commercially successful learning techniques ever developed. Among many other applications, LSTMs have been used in Google Allo [39], Google Translate [84], Apple's Siri [46], and Amazon Alexa [25]. However, recently, the dominance of LSTM may have begun to wane. ResNet has been shown to have theoretical advantages over LSTM, and it outperforms LSTM in a wide range of applications [63].

Figure 20 illustrates an LSTM. The obvious difference from a generic vanilla RNN is that an LSTM has two lines entering and exiting each state. As in a standard RNN, one of these lines represents the hidden state, while the second line is designed to serve as a gradient "highway" during backpropagation. In this way, the gradient can "flow" much further back with less chance that it will vanish along the way.

In Fig. 21, we expand one of the LSTM cells L_t that appear in Fig. 20. Here, σ is the sigmoid function, τ is the hyperbolic tangent (i.e., tanh) function, the operators "$\times$" and "$+$" are pointwise multiplication and addition, respectively, while "$\|$" indicates concatenation of vectors. The vector i_t is the "input" gate, f_t is the "forget" gate, and o_t is the "output" gate. The vector g_t is an intermediate gate and does not have a cool name, but is sometimes referred to as the "gate" gate [47], which, come to think of it, is especially cool. We have much more to say about these gates below.

The gate vectors that appear in Fig. 21 are computed as

$$f_t = \sigma\left(W_f\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_f\right) \qquad g_t = \tau\left(W_g\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_g\right)$$

$$i_t = \sigma\left(W_i\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_i\right) \qquad o_t = \sigma\left(W_o\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_o\right),$$

while the outputs are

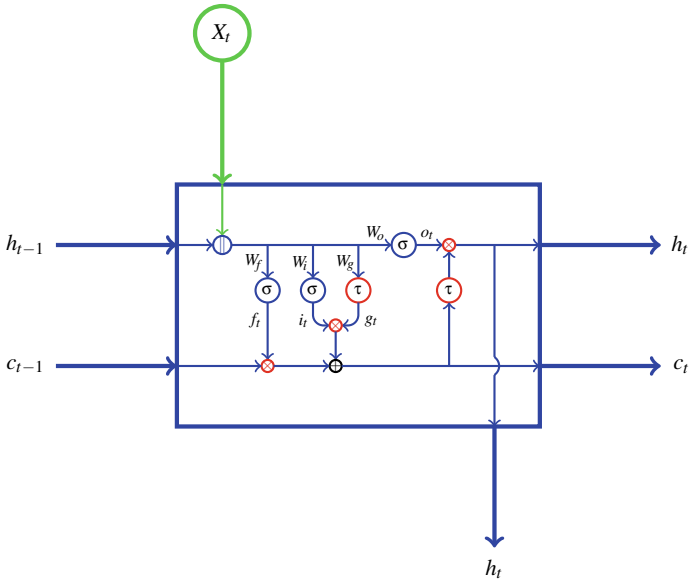


Fig. 21 One timestep of an LSTM

$$c_t = f_t \otimes c_{t-1} \oplus i_t \otimes g_t$$
$$h_t = o_t \otimes \tau(c_t),$$

where "$\otimes$" is pointwise multiplication and "$\oplus$" is the usual pointwise addition. Note that each of the weight matrices is $n \times 2n$.

In matrix form, ignoring the bias terms b, we have

$$\begin{pmatrix} i_t \\ f_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tau \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix},$$

where X_t and h_{t-1} are column vectors of length n, and W is the $4n \times 2n$ weight matrix

$$W = \begin{pmatrix} W_i \\ W_f \\ W_o \\ W_g \end{pmatrix}$$

Further, each of the gates i_t, f_t, o_t, and g_t is a column vectors of length n. Recall that the sigmoid σ squashes its input to be within the range of 0 to 1, whereas the tanh function τ gives output within the range of -1 to $+1$.

To highlight the intuition behind LSTM, we follow a similar approach as that given in the excellent presentation [47]. Specifically, we focus on the extreme cases, that is, we assume that the output of each sigmoid σ is either 0 or 1, and each hyperbolic tangent τ is either -1 or $+1$. Then the forget gate f_t is a vector of 0s and 1s, where the 0s tell us the elements of c_{t-1} that we forget and the 1s indicate the elements to remember. In the middle section of the diagram, the input gate i_t and gate gate g_t together determine which elements of c_{t-1} to increment or decrement. Specifically, when element j of i_t is 1 and element j of g_t is $+1$, we increment element j of c_{t-1}. And if element j of i_t is 1 and element j of g_t is -1, then we decrement element j of c_{t-1}. This serves to emphasize or de-emphasize particular elements in the new-and-improved cell state c_t. Finally, the output gate o_t determines which elements of the cell state will become part of the hidden state h_t. Note that the hidden states h_t is fed into the output layer of the LSTM. Also note that before the cell states are operated on by the output gate, the values are first squeezed down to be within the range of $-1-+1$ by the τ function.

Of course, in general, the LSTM gates are not simply countered that increment or decrement by 1. But, the intuition is the same, that is, the gates keep track of incremental changes thus allowing relevant information to flow over long distances via the cell state. In this way, LSTM negates some of the limitations caused by vanishing gradients.

7.3 Gated Recurrent Units

As mentioned above, there are a large number of variants of the basic LSTM architecture. Most such variants are slight variants, with only minor changes from a standard LSTM. A gated recurrent unit (GRU), on the other hand, is a fairly radical departure from an LSTM. Although the internal state of a GRU is somewhat complex and, perhaps, less intuitive than that of an LSTM, there are fewer parameters in a GRU, and hence it is easier to train a GRU, and less training data is required. The wiring diagram for a GRU is given in Fig. 22.

The gate vectors that appear in Fig. 21 are computed as

$$z_t = \sigma\left(W_z\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_z\right)$$

$$r_t = \sigma\left(W_r\begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + b_r\right)$$

$$g_t = \tau\left(W_g\begin{pmatrix} r_t \otimes h_{t-1} \\ X_t \end{pmatrix} + b_g\right),$$

while the output is

$$h_t = (1 - z_t) \otimes h_{t-1} \oplus z_t \otimes g_t,$$

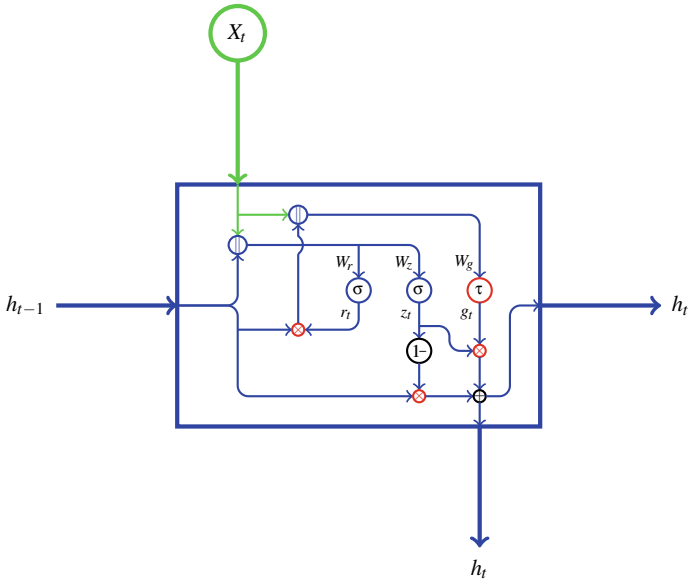


Fig. 22 One timestep of a GRU

where "$\otimes$" is pointwise multiplication and "$\oplus$" is the usual pointwise addition. Note that each of the weight matrices is $n \times 2n$.

In matrix form, ignoring the bias terms b, we have

$$\begin{pmatrix} z_t \\ r_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ 0 \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ X_t \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \tau \end{pmatrix} W \begin{pmatrix} r_t \otimes h_{t-1} \\ X_t \end{pmatrix},$$

where X_t and h_{t-1} are column vectors of length n, and W is the $3n \times 2n$ weight matrix

$$W = \begin{pmatrix} W_z \\ W_r \\ W_g \end{pmatrix}$$

Each of the gates z_t, r_t, and g_t is a column vectors of length n.

The intuition behind a GRU is that it replaces the input, forget, and output gates of an LSTM with just two gates—an "update" gate z_t and a "reset" gate r_t. The GRU update gate serves a similar purpose as the combined output and forget gates of an LSTM. Specifically, the update serves to determine what to output (or write) and what to forget. The function $1 - z_t$ in the GRU implies that anything that is not output must be forgotten. Thus, the GRU is less flexible as compared to an LSTM since an LSTM allows us to independently select elements for output and elements that are

forgotten. The GRU reset gate and the LSTM input gate each serve to combine new input with previous memory.

The gating in a GRU is more complex and somewhat less intuitive as compared to that found in an LSTM. In any case, the most radical departure of the GRU from the LSTM architecture is that there is no cell state in a GRU. This implies that any memory must be stored in the hidden state h_t. This simplification (as compared to an LSTM) relies on the fact that in a GRU, the write and forget operations have been combined.

7.4 Recursive Neural Network

We mention in passing that recursive neural networks can be viewed as generalizing recurrent neural networks.[7] In a recursive neural network, we can recurse over any hierarchical structure, with trees being the archetypal example. Then training can be accomplished via backpropagation through structure (BPTS), often using stochastic gradient descent for simplicity. In contrast, a recurrent neural network is restricted to one particular structure—that of a linear chain.

7.5 Last Word on RNNs

RNNs are useful in cases where the input data is sequential. Generic RNN architectures are subject to vanishing and exploding gradients, which limit the length of the history (or gaps) that can effectively be incorporated into such models. Relatively complex RNN-based architectures—such as LSTM and its variants—have been developed that can better handle such gradient issues. These architectures have proven to be commercially successful across a wide range of products.

A good general discussion of RNNs can be found in [59], and an overview of various RNN-specific topics—with links to many relevant articles—is available at [58]. A more detailed (mathematical) description can be found in Chap. 10 of [20]. The slides at [47] provide a good general introduction to RNNs, with nice examples and a brief, but excellent, discussion of LSTM.

[7] Unfortunately, "recursive neural network" is typically also abbreviated as RNN. Here, we reserve RNN for recurrent neural networks and we do not use any abbreviation when referring to recursive neural networks.

7.6 RNNs in Malware Analysis

In a commercial sense, LSTMs are surely the most successful deep learning technique yet developed, so it is not surprising that LSTMs have been successfully applied to the malware detection problem [50]. Both LSTMs and GRUs—along with CNNs—are considered in [2], with the authors claiming a major improvement over relevant previous work. The paper [31] considers an adversarial attack, where the attacker can defeat a system that uses RNNs based on API calls.

There are many applications of RNNs in areas of information security outside of the malware domain. In [87], CNN and LSTM architectures are used to detect cybersecurity events, based on social networking messages. Other infosec applications of LSTMs include generating security ontologies [19], network security [49], breaking CAPTCHAs [11], host-based intrusion detection [41], and network anomaly detection [13], among others.

8 Residual Networks

At the time of this writing, residual network (ResNet) is considered the state of the art in deep learning for many image analysis problems. A residual network is one in which instead of approximating a function $F(x)$, we approximate the "residual," which is defined as $H(x) = F(x) - x$. Then the desired solution is given by $F(x) = H(x) + x$.

The original motivation for considering residuals was based on the observation that deeper networks sometimes produce worse results, even when vanishing gradients are not the cause [27]. This is somewhat counterintuitive, as the network should simply learn identity mappings when a model is deeper than necessary. To overcome this "degradation" problem, the authors of [27] experiment with residual mappings and provide extensive empirical evidence that the resulting ResNet architecture yields improved results as compared to standard feedforward networks for a variety of problems. The authors of [27] conjecture that the success of ResNet follows from the fact that the identity map corresponds to a residual of zero, and "if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers."

Whereas LSTM uses a complex gating structure to ease gradient flow, ResNet defines additional connections that correspond to identity layers. This enables ResNet to deal with vanishing gradients, as well as the aforementioned degradation problem. These identity layers allow a ResNet model to skip over layers during training, which serves to effectively reduce the minimum depth when training. Intuitively, ResNet is able to train deeper networks by, in effect, training over a considerably shallower network in the initial stages, with later stages of training serving to flesh out the intermediate connections. This approach was inspired by pyramidal cells in

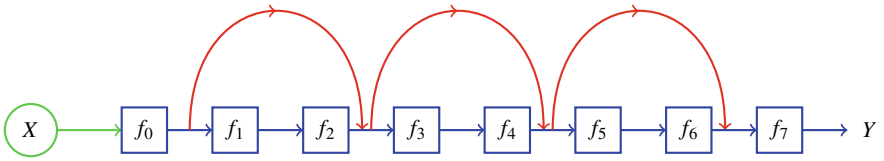


Fig. 23 Example of a ResNet architecture

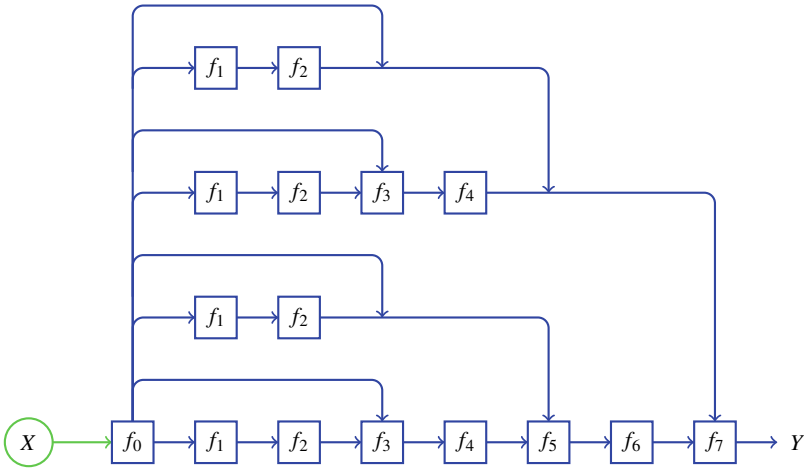


Fig. 24 Another view of the ResNet architecture in Fig. 23

the brain, which have a similar characteristic in the sense that they bridge "layers" of neurons [76].

A very high-level illustrative example of a ResNet architecture is given in Fig. 23, where each curved edge represents an identity transformation. Note that in this case, the identity transformations enable the model to skip over two layers. In principle, ResNet would seem to be applicable to any flavor of deep neural network, but in practice, it seems to applied to CNNs.

If a ResNet has N identity paths, then the network contains 2^N distinct feedforward networks. For example, the ResNet in Fig. 23 can be expanded into the graph in Fig. 24. Note that most of the paths in a ResNet are relatively short.

Surprisingly, the paper [81] provides evidence that in spite of being trained simultaneously, the multiple paths in a ResNet "show ensemble-like behavior in the sense that they do not strongly depend on each other." And perhaps an even more surprising result in [81] shows that "only the short paths are needed during training, as longer paths do not contribute any gradient." In other words, a deep ResNet architecture is more properly viewed as a collection of multiple, relatively shallow networks.

8.1 ResNet in Malware Analysis

At the time of this writing, ResNet is a relative newcomer and the level of research in the security domain is somewhat limited. Nevertheless, ResNet architectures have shown promise for dealing with the usual suspects, namely malware analysis [40, 66] and intrusion detection [43, 83].

9 Generative Adversarial Network

Let $\{X_i\}$ be a collection of samples and $\{Y_i\}$ a corresponding set of class labels. In statistics, a *discriminative* model is one that models the conditional probability distribution $P(Y \mid X)$. Such a discriminative model can be used to classify samples—given an input X of the same type as the training samples $\{X_i\}$, the model enables us to easily determine the most likely class of X by simply computing $P(Y \mid X)$ for each class label Y.

In contrast, a model is said to be *generative* if it models the joint probability distribution of X and Y, which we denote as $P(X, Y)$. Such a model is called "generative" because, by sampling from this distribution, we can generate new pairs (X_i, Y_i) that fit the probability distribution. Note that we can produce a discriminative model from a generative model, since

$$P(Y \mid X) = \frac{P(X, Y)}{P(X)}.$$

Therefore, in some sense, a generative model is inherently more general than a discriminative model.

Consider, for example, hidden Markov models (HMM) [77], which are a popular class of classic machine learning techniques. An HMM is defined by the three matrices in $\lambda = (A, B, \pi)$, where π is the initial state distribution, A contains the transition probability distributions for the hidden states, and B consists of the observation probability distributions corresponding to the hidden states. If we train an HMM on a given dataset, then we can easily generate samples that match the probability distributions of the HMM. To generate such samples, we first randomly select an initial state based on the probabilities in π. Then we repeat the following steps until the desired observation sequence length is reached: Randomly select an observation based on the current state, using the probabilities in B, and randomly select the next state, based on the probabilities in A. The resulting observation sequence will be indistinguishable (in the HMM sense) from the data that was used to train the HMM.

From the discussion in the previous paragraph, it is clear that a trained HMM is a generative model. However, it is more typical to use an HMM as a discriminative model. In discriminative mode, we determine a threshold, then we classify a given observation sequence as matching the model if its HMM score is above the specified

threshold. This example shows that in practice, it is easy to use a generative model as a discriminative model.

On the other hand, while a trained SVM serves to classify samples, we could not use such a model to generate samples that match the training set. Thus, an SVM is an example of a discriminative model.

In the realm of deep learning, a discriminative network is designed to classify samples, while a generative network is designed to generate samples that "fit" the training data. From the discussion above, it is clear that we can always obtain a discriminative model from a generative model. Intuitively, it would seem that training a (more general) generative model in order to obtain a (more specific) discriminative model would be undesirable since we do not need the full generality of the model. However, reality appears to be somewhat more subtle. In [60], it is shown that for one generative–discriminative pair (naïve Bayes and logistic regression) the discriminative models do indeed have a lower asymptotic error; however, the generative models consistently converge faster. This suggests that with limited training data, a generative model might produce a superior discriminative model, as compared to directly training the corresponding discriminative model. In any case, in the realm of deep learning, discriminative models dominate, with an example of a typical application being image classification. In contrast, generative models have only recently come into vogue, with an example application being the creation of fake images.

Now, suppose that when training a discriminative neural network, in addition to the real training data, we generate "fake" training samples that follow a similar probability distribution as the real samples. Further, suppose that these fake training samples are designed to trick the discriminative network into making classification mistakes. Such samples would tend to improve the training of the network, thus making it stronger and more effective than if we had restricted the training to only the real data.

Although intuitively appealing, several problems arise when trying to implement a training technique based on fake samples. For one thing, we generally do not know the distribution of the training set, which often lives in an extremely high dimensional space of great complexity. Another issue is that during training, the discriminative network is constantly evolving, so determining samples that are likely to trick the network is a moving target. Another concern is that if the fake training samples are too difficult—or too easy—to distinguish at any point in the training process, we are unlikely to see any improvement over simply using the real training data

Several techniques have been proposed to try to take advantage of fake data so as to improve the training process. In the case of a generative adversarial network (GAN), we use a neural network to generate the fake data—a generative network is trained to defeat a discriminative network. Furthermore, the discriminative and generative networks are trained simultaneously in a minimax game. This approach sidesteps the complications involved in trying to model the probability distribution of the training samples. In fact, the generative network in a GAN simply uses random noise as its underlying probability distribution.

To summarize, a GAN consists of two competing neural networks—a generative network and a discriminative network—with the generative network creating fake

data that is designed to defeat the discriminative network. The two networks are trained simultaneously following a game-theoretic approach. In this way, both networks improve, with the ultimate objective being a discriminative model (and/or a generative model) that is stronger than it would have been if it was trained only on the real training data.

We define two neural networks, namely a discriminator $D(x;\theta_d)$, and a generator $G(z;\theta_g)$, where θ_d consists of the parameters of the discriminator network, and θ_g consists of the parameters of the generator network. Here, we describe the training process in terms of images, but other types of data could be used. Also, to simplify the notation, we suppress the dependence on θ_d and θ_g in the remainder of this discussion, except where it is essential for understanding and may not be clear from context.

The generator $G(z)$ produces a fake image (based on the random seed value z) with the goal of tricking the discriminator into believing it is a real training image. In contrast, the discriminator $D(x)$ returns a value in the range of 0– 1 that can be viewed as its estimate of the probability that the image x is real. For example, $D(x) = 1$ means that the discriminator is completely certain that the image is real, while $D(x) = 0$ tells us that the discriminator is sure that the image is fake, and $D(x) = 1/2$ implies that the discriminator is clueless. Note that the discriminator must deal with both real and fake images, while the generator is only concerned with generating fake images that trick the discriminator.

The generator G wins if D thinks its fake images are real. Thus, we can train G by making $1 - D(G(z))$ as close to zero as possible or, equivalently, by minimizing $\log(1 - D(G(z)))$. On the other hand, D wins if it can distinguish the fake images from real images so, ideally, when training D, we want $D(x) = 1$, when x is a real image, and $D(G(z)) = 0$ for fake images $G(z)$. Therefore, we can train D by maximizing $D(x)(1 - D(G(z))$ or, equivalently, by maximizing $\log(D(x)) + \log(1 - D(G(z)))$. We want the D and G models to be in competition, so they can strengthen each other. This can be accomplished by formulating the training in terms of the minimax game

$$\min_G \max_D \Big(E\big(\log(D(x))\big) + E\big(\log(1 - D(G(z)))\big)\Big), \tag{2}$$

where E is the expected value, relative to the implied probability distribution. Specifically, for the max over D, the expectation is with respect to the real sample distribution which has parameters θ_d, while for the min over G, the expectation is with respect to the fake sample distribution, which is specified by the parameters θ_g.

In the case of stochastic gradient descent (or ascent), at each iteration, we consider one real sample x and one fake sample $G(z)$. Then, due to the max in equation (2), we first perform gradient ascent to update the discriminator network D. This is followed by gradient descent to update generator network G. Of course, both of these steps rely on backpropagation.

Note that for the discriminator network D, the backpropagation error term involves

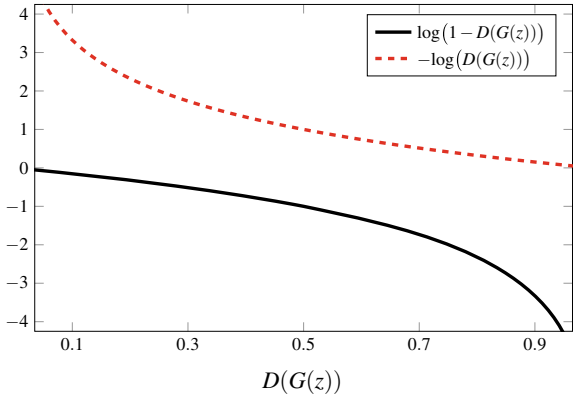


Fig. 25 Gradient of generator network G

$$\log\big(D(x)\big) + \log\big(1 - D(G(z))\big),$$

while for the generator network G, the error term involves only

$$\log\big(1 - D(G(z))\big). \tag{3}$$

Of course, in practice, we would typically use a minibatch of, say, m real samples and m fake samples at each update of D and G, rather than a strict stochastic gradient descent/ascent.

There is one technical issue that arises when attempting to train the generator network G as outlined above. As illustrated in Fig. 25, the gradient of the expression in (3) is nearly flat for values of $D(G(z))$ near zero. This implies that, early in training, when the generator network is sure to be extremely weak—and hence the discriminator can easily identify most $G(z)$ images as fake—it will be difficult for the G network to learn. From, Fig. 25, we also see that

$$\log\big(D(G(z))\big) \tag{4}$$

is relatively steep near zero. Hence, instead training G based on a gradient ascent involving equation (3), we perform gradient *descent* based on (4). Note that we have simply replaced the problem of maximizing $1 - D(G(z))$ with the equivalent problem of minimizing the probability $D(G(z))$.

The algorithm for training a GAN is summarized in Fig. 26. In some applications, letting `iters` $= 1$ works best, while in others, `iters` > 1 yields better results. In the latter case, we update the discriminator network D multiple times for each update of the generator network G. This implies that in such cases, the generator might otherwise overwhelm the discriminator, that is, the generator is in some sense easier to train. Finally, while a GAN certainly is an advanced architecture, it is important to realize that training reduces to a fairly straightforward application of gradient ascent.

0: initialize parameters θ_d and θ_g and `iters` ≥ 1
1: repeat
2: for $k = 1$ to `iters`
3: randomly select n noise samples $Z = (z_0, z_1, \ldots, z_{n-1})$
4: randomly select n real samples $X = (x_0, x_1, \ldots, x_{n-1})$
5: update θ_d by gradient ascent on
$$\sum_{i=0}^{n-1} \Big(\log\big(D(x_i)\big) + \log\big(1 - D(G(z_i))\big) \Big)$$
6: next k
7: randomly select n noise samples $Z = (z_0, z_1, \ldots, z_{n-1})$
8: update θ_g by gradient ascent on
$$\sum_{i=0}^{n-1} \log\big(D(G(z_i))\big)$$
9: until stopping criteria is met
10: return(θ_d, θ_g)

Fig. 26 GAN training algorithm

As with LSTM, there are a vast number of variations on the basic GAN approach outlined here; see [48] for a list of nearly 50 such variants. Additional sources of information on GANs include the original paper on the subject [21] and the excellent slides at [48].

9.1 GANs in Malware Analysis

GANs seem to show promise for dealing with some of the most challenging problems in information security. For example, GANs have been applied with some success to zero-day malware detection [32, 42]. In addition, the generative aspect of a GAN can be used to create challenging security problems in the "lab," thus enabling researchers to consider defenses against potential threats before those threats arise in a real-world setting [67].

10 Extreme Learning Machines

As with most aspects of ELMs, the origin of the technique is somewhat controversial. The unfortunate terminology of "Extreme Learning Machine" was apparently first used in [24]. Regardless of the origin of the technique, ELMs are essentially randomized feedforward neural networks that effectively minimize the cost of training.

An ELM consists of a single layer of hidden nodes, where the weights between inputs and hidden nodes are randomly initialized and remain unchanged throughout training. The weights that connect the hidden nodes to the output are trained, but due to the simple structure of an ELM, these weights can be determined by solving

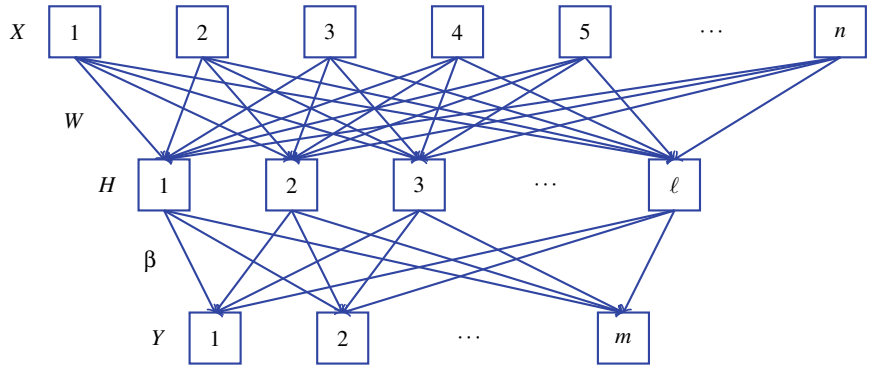


Fig. 27 Architecture of an ELM model

linear equations—more precisely, by solving a linear regression problem. Since no backpropagation is required, ELMs are far more efficient to train, as compared to other neural network architectures. However, since the weights in the hidden layer are not optimized, we will typically require more weights in an ELM, which implies that the testing phase may be somewhat more costly, as compared to a network trained by backpropagation. Nevertheless, in applications where models must be trained frequently, ELMs can be competitive.

Consider the ELM architecture shown in Fig. 27, where X denotes the input layer, H is the hidden layer, and Y is the output layer. In this example, there are N samples of the form (x_i, y_i) for $i = 1, 2, \ldots, N$, where $x_i = \begin{pmatrix} x_{i_1} & x_{i_2} & \ldots & x_{i_n} \end{pmatrix}^T$ is the feature vector for sample i and $y_i = \begin{pmatrix} y_{i_1} & y_{i_2} & \ldots & y_{i_m} \end{pmatrix}^T$ are the output labels, where T indicates the transposition operation. Then the input and output for the ELM are as $X = \begin{pmatrix} x_1 & x_2 & \ldots & x_n \end{pmatrix}^T$ and $Y = \begin{pmatrix} y_1 & y_2 & \ldots & y_m \end{pmatrix}^T$, respectively. In this example, the hidden layer H has ℓ neurons. We denote the activation function of the hidden layer as $g(x)$.

To train an ELM, we randomly select the weight matrix that connects the input layer X to the hidden layer H. We denote this randomly assigned weight matrix as $W = \begin{pmatrix} w_1 & w_2 & \ldots & w_\ell \end{pmatrix}$, where each w_i is a column vector. We also randomly select the bias matrix $B = \begin{pmatrix} b_1 & b_2 & \ldots & b_\ell \end{pmatrix}$ for this same layer. During the training phase, both W and B remain unchanged.

After W and B have been initialized, the output of the hidden layer H is given by

$$H = g(WX + B).$$

The output of the ELM is denoted as Y and is calculated as

$$Y = H\beta,$$

where β is the weight matrix for the output layer.

The values of the weights β at the hidden layer are learned via linear least squares, and can be computed using $H^\dagger$, the Moore–Penrose generalized inverse of H, as discussed below. It is worth emphasizing that the only parameters that are learned in the ELM are the elements of β.

Given that Y is the desired output, a unique solution of the system based on least squared error can be found as follows. We denote the Moore–Penrose generalization inverse of H as $H^\dagger$, which is defined as

$$H^\dagger = \begin{cases} (H^T H)^{-1} H^T \text{ if } H^T H \text{ is nonsingular} \\ H^T (H H^T)^{-1} \text{ if } H H^T \text{ is nonsingular} \end{cases}$$

Then the desired solution β is given by

$$\beta = H^\dagger Y.$$

After calculating β, the training phase ends. For each test sample x, the output Y can be calculated as

$$Y = g\big(C(x)\big)\beta,$$

where $C(x)$ is defined below. The entire training process is extremely efficient, particularly in comparison to the backpropagation technique that is typically used to train neural networks [80].

For the research reported in this paper, we use the Python implementation of ELMs given in [17]. This implementation uses input activations that are a weighted combination of two functions referred to as an "MLP" kernel and an "RBF" kernel—we employ the same terminology here. The MLP kernel is simply the linear operation

$$M(x) = Wx + B,$$

where the weights W and biases B are randomly selected from a normal distribution. This is the kernel function that is typically associated with a standard ELM.

The RBF kernel is considerably more complex and is based on generalized radial basis functions as defined in [18]. The details of this RBF kernel go beyond the scope of this paper; see [18] for additional information and, in particular, examples where this kernel is applied to train ELMs. We use the notation $R(x)$ to represent the RBF kernel. Also, it is worth noting that the RBF kernel is much more costly to compute, and hence its use does somewhat negate one of the major advantages of an ELM.

The input activations are given by

$$C(x) = \alpha M(x) + (1 - \alpha) R(x), \tag{5}$$

where $0 \le \alpha \le 1$ is a user-specified mixing parameter. Note that for $\alpha = 0$ we use only the MLP kernel $M(x)$ and for $\alpha = 1$, only the RBF kernel $R(x)$ is used.

10.1 ELMs in Malware Analysis

In [34], ELMs are compared to CNNs for malware classification, and it is shown that ELMs can outperform CNNs in some cases. This is impressive since ELMs have training times that are only a small fraction of those required for comparable CNNs. ELMs have also been applied to malware detection on the Android platform in [90], where the training is based on static features, with reasonably strong results. In [71], the authors consider the effectiveness of a technique that they refer to as high-performance extreme learning machines (HP-ELM). By varying the features and activation functions of their HP-ELM architecture, they achieve high accuracy on a challenging dataset. A two-layer ELM is applied to the malware detection problem in [33]. A partially connected network is used between the input and the first hidden layer, and this layer is aggregated with a fully connected network in the second layer. The authors utilize an ensemble to improve the accuracy and robustness of the resulting ELM-based system.

11 Word Embedding Techniques

Word2Vec is a technique for embedding terms in a high-dimensional space, where the term embeddings are obtained by training a shallow neural network. After the training process, words that are more similar in context will tend to be closer together in the Word2Vec space.

Perhaps surprisingly, meaningful algebraic properties also hold for Word2Vec embeddings. For example, according to [53], if we let

$$w_0 = \text{“king”}, w_1 = \text{“man”}, w_2 = \text{“woman”}, w_3 = \text{“queen”}$$

and $V(w_i)$ is the Word2Vec embedding of word w_i, then $V(w_3)$ is the vector that is closest—in terms of cosine similarity—to

$$V(w_0) - V(w_1) + V(w_2).$$

Results such as this indicate that Word2Vec embeddings capture significant aspects of the semantics of the language.

The focus of this section is Word2Vec, but before discussing this popular and effective word embedding technique, we consider a couple of alternatives. First, we discuss simple embedding strategies based on hidden Markov models. Then we briefly consider a word embedding technique the uses PCA. Finally, we discuss the main ideas behind Word2Vec.

11.1 HMM2Vec

To begin, we consider individual letter embeddings, as opposed to word embeddings. We call the letter embedding technique considered here Letter2Vec.

Recall that an HMM is defined by the three matrices A, B, and π, and is denoted as $\lambda = (A, B, \pi)$. The π matrix contains the initial state probabilities, A contains the hidden state transition probabilities, and B consists of the observation probability distributions corresponding to the hidden states. Each of these matrices is row stochastic, that is, each row satisfies the requirements of a discrete probability distribution. Notation-wise, we let N be the number of hidden states, M is the number of distinct observation symbols, and T is the length of the observation (i.e., training) sequence. Note that M and T are determined by the training data, while N is a user-defined parameter. For more details in HMMs, see [77] or Rabiner's fine paper [65].

Suppose that we train an HMM on a sequence of letters extracted from English text, where we convert all uppercase letters to lowercase and discard any character that is not an alphabetic letter or word-space. Then $M = 27$, and we select $N = 2$ hidden states, and we use $T = 50{,}000$ observations for training. Note that each observation is one of the $M = 27$ symbols (letters plus word-space). For the example discussed below, the sequence of $T = 50{,}000$ observations was obtained from the Brown corpus of English [7]. Of course, any source of English text could be used.

For one specific case, an HMM trained with the parameters listed in the previous paragraph yields the B matrix in Table 1. Observe that this B matrix gives us two probability distributions over the observation symbols—one for each of the hidden states. We observe that one hidden state essentially corresponds to vowels, while the other corresponds to consonants. This simple example nicely illustrates the concept of machine learning, as no a priori assumption was made concerning consonants and vowels, and the only parameter we selected was the number of hidden states N. Through the training process, the model learned a crucial aspect of English directly from the data. This illustrative example is discussed in more detail in [77] and originally appeared in Cave and Neuwirth's classic paper [8].

Suppose that for a given letter ℓ, we define its Letter2Vec representation $V(\ell)$ to be the corresponding row of the matrix B^T in Table 1. Then, for example,

$$\begin{aligned} V(\text{a}) &= \begin{pmatrix} 0.13537 & 0.00364 \end{pmatrix} \quad V(\text{e}) = \begin{pmatrix} 0.21176 & 0.00223 \end{pmatrix} \\ V(\text{s}) &= \begin{pmatrix} 0.00032 & 0.11069 \end{pmatrix} \quad V(\text{t}) = \begin{pmatrix} 0.00158 & 0.15238 \end{pmatrix} \end{aligned} \tag{6}$$

Next, we consider the distance between these Letter2Vec representations. Instead of using Euclidean distance, we measure the cosine similarity.[8]

The cosine similarity of vectors X and Y is the cosine of the angle between the two vectors. Let $S(X, Y)$ denote the cosine similarity between vectors X and Y. Then for $X = (X_0, X_1, \ldots, X_{n-1})$ and $Y = (Y_0, Y_1, \ldots, Y_{n-1})$,

[8] Cosine similarity is not a true metric, since it does not, in general, satisfy the triangle inequality.

Table 1 Final B^T for HMM

Letter	State 0	State 1	Letter	State 0	State 1
a	0.13537	0.00364	n	0.00035	0.11429
b	0.00023	0.02307	o	0.13081	0.00143
c	0.00039	0.05605	p	0.00073	0.03637
d	0.00025	0.06873	q	0.00019	0.00134
e	0.21176	0.00223	r	0.00041	0.10128
f	0.00018	0.03556	s	0.00032	0.11069
g	0.00041	0.02751	t	0.00158	0.15238
h	0.00526	0.06808	u	0.04352	0.00098
i	0.12193	0.00077	v	0.00019	0.01608
j	0.00014	0.00326	w	0.00017	0.02301
k	0.00112	0.00759	x	0.00030	0.00426
l	0.00143	0.07227	y	0.00028	0.02542
m	0.00027	0.03897	z	0.00017	0.00100
Space	0.34226	0.00375	–	–	–

$$S(X,Y) = \frac{\sum_{i=0}^{n-1} X_i Y_i}{\sqrt{\sum_{i=0}^{n-1} X_i^2}\sqrt{\sum_{i=0}^{n-1} Y_i^2}}$$

In general, we have $-1 \le S(X,Y) \le 1$, but since our Letter2Vec encoding vectors consist of probabilities—and hence are non-negative values—we always have $0 \le S(X,Y) \le 1$.

When considering cosine similarity, the length of the vectors is irrelevant, as we are only considering the angle between vectors. Consequently, we might want to consider vectors of length one, $\widetilde{X} = X/\|X\|$ and $\widetilde{Y} = Y/\|Y\|$, in which case the cosine similarity simplifies to the dot product

$$S(\widetilde{X},\widetilde{Y}) = \sum_{i=0}^{n-1} \widetilde{X}_i \widetilde{Y}_i$$

Henceforth, we use the notation $\widetilde{X}$ to indicate a vector X that has been normalized to be of length one.

For the vector encodings in (6), we find that for the vowels "a" and "e", the cosine similarity is $S(V(\text{a}), V(\text{e})) = 0.9999$. In contrast, the cosine similarity of the vowel "a" and the consonant "t" is $S(V(\text{a}), V(\text{t})) = 0.0372$. The normalized vectors $V(\text{a})$

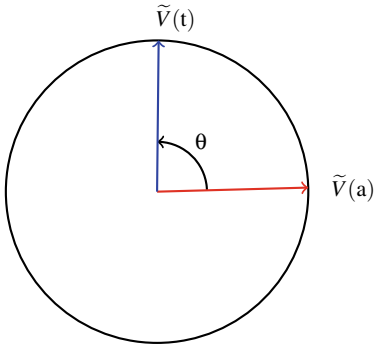


Fig. 28 Normalized vectors $\widetilde{V}(\mathrm{a})$ and $\widetilde{V}(\mathrm{t})$

and $V(\mathrm{t})$ are illustrated in Fig. 28. Using the notation in this figure, cosine similarity is $S(V(\mathrm{a}), V(\mathrm{t})) = \cos(\theta)$.

These results indicate that these Letter2Vec encodings—which are derived from a trained HMM—provide useful information on the similarity (or not) of pairs of letters. Note that we could obtain a vector encoding of any dimension by simply training an HMM with the number of hidden states N equal to the desired dimension.

Our HMM-based approach to Letter2Vec encoding is interesting, but we want to encode words, not letters. Analogous to the Letter2Vec embeddings discussed above, we could train an HMM on words and then use the columns of the resulting B matrix (equivalently, the rows of B^T) to define word vectors. The state of the art for Word2Vec uses a dataset corresponding to $M = 10{,}000$, $N = 300$ and $T = 10^9$. Training an HMM with similar parameters would be decidedly non-trivial, as the work factor is on the order of N^2T.

While the word embedding technique discussed in the previous paragraph—we call it HMM2Vec—is plausible, it has some potential limitations. Perhaps the biggest issue with HMM2Vec is that we typically train an HMM based on a Markov model of order one. This means that the current state only depends on the immediately preceding state. By basing our word embeddings on such a model, the resulting vectors would likely provide only a very limited sense of context. While we can train HMMs using models of a higher order, the work factor would be prohibitive.

11.2 PCA2Vec

Another option for generating embedding vectors is to apply PCA to a matrix of pointwise mutual information (PMI). To construct a PMI matrix, based on a specified window size W, we compute the probabilities $P(w_i, w_j)$ for all pairs of words (w_i, w_j) that occur within a window W of each other within dataset, and we also compute $P(w_i)$ for each individual word w_i. Then we define the PMI matrix as $X = \{x_{ij}\}$ as

$$x_{ij} = \log\left(\frac{P(w_j, w_i)}{P(w_i)P(w_j)}\right) = \log P(w_j, w_i) - \log P(w_i) - \log P(w_j).$$

Let X_i be column i of X. We use X_i as the feature vector for word w_i and perform PCA (using SVD) based on these X_i feature vectors. As usual, we project the feature vectors X_i onto the resulting eigenspace. Finally, by choosing the N dominant eigenvalues for this projection, we obtain word embedding vectors of length N.

It is shown in [56] that these embedding vectors have many similar properties as Word2Vec embeddings, with the author providing examples analogous to those we give in the next section. Interestingly, it may be beneficial in some applications to omit a few of the dominant eigenvectors when determining the PCA2Vec embedding vectors [45].

For more details on using PCA to generate word embeddings, see [45]. The aforecited blog [56] gives an intuitive introduction to the topic.

11.3 Word2Vec

Word2Vec uses a similar approach as the HMM2Vec concept outlined above. But, instead of using an HMM, Word2Vec is based on a shallow (one hidden layer) neural network. Analogous to HMM2Vec, in Word2Vec, we are not interested in the resulting model itself, but instead we make use the learning that is represented by the trained model to define word embeddings. Next, we consider the basic ideas behind Word2Vec. Our presentation is fairly similar to that found in the excellent tutorial [51].

Suppose that we have a vocabulary of size M. We encode each word as a "one-hot" vector of length M. For example, suppose that our vocabulary consists of the set of $M = 8$ words

$$\begin{aligned} W &= (w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7) \\ &= (\text{“for”}, \text{“giant”}, \text{“leap”}, \text{“man”}, \text{“mankind”}, \text{“one”}, \text{“small”}, \text{“step”}) \end{aligned}$$

Then we encode "for" and "man" as

$$E(w_0) = E(\text{“for”}) = 10000000 \text{ and } E(w_3) = E(\text{“man”}) = 00010000,$$

respectively.

Now, suppose that our training data consists of the phrase

$$\text{“one small step for man one giant leap for mankind”}. \tag{7}$$

Table 2 Training data

Offset	Training pairs
" one small step . . ."	(one, small), (one, step)
"one small step for . . ."	(small, one), (small, step), (small, for)
"one small step for man . . ."	(step, one), (step, small), (step, for), (step, man)
". . . small step for man one . . ."	(for, small), (for, step), (for, man), (for, one)
". . . step for man one giant . . ."	(man, step), (man, for), (man, one), (man, giant)
". . . for man one giant leap . . ."	(one, for), (one, man), (one, giant), (one, leap)
". . . man one giant leap for . . ."	(giant, man), (giant, one), (giant, leap), (giant, for)
". . . one giant leap for mankind"	(leap, one), (leap, giant), (leap, for), (leap, mankind)
". . . giant leap for mankind"	(for, giant), (for, leap), (for, mankind)
". . . leap for mankind "	(mankind, leap), (mankind, for)

To obtain training samples, we specify the window size, and for each offset, we use all pairs of words within the specified window. For example, if we select a window size of two, then from (7), we obtain the training pairs in Table 2.

Consider the pair "(for,man)" from the fourth row in Table 2. As one-hot vectors, this training pair corresponds to input 10000000 and output 00010000.

A neural network similar to that in Fig. 29 is used to generate Word2Vec embeddings. The input is a one-hot vector of length M representing the first element of a training pair, such as those in Table 2, and the network is trained to output the second element of the ordered pair. The hidden layer consists of N linear neurons and the output layer uses a softmax function to generate M probabilities, where p_i is the probability of the output vector corresponding to w_i for the given input.

Observe that the Word2Vec network in Fig. 29 has NM weights that are to be determined, as represented by the blue lines from the hidden layer to the output layer. For each output node ω_i, there are N edges (i.e., weights) from the hidden layer. The N weights that connect to output node ω_i form the Word2Vec embedding $V(w_i)$ of the word w_i.

As mentioned above, the state of the art in Word2Vec for English text is based on a vocabulary of $M = 10{,}000$ words, and embedding vectors of length $N = 300$. These embeddings are obtained by training on a set of about 10^9 samples. Clearly, training a model of this magnitude is an extremely challenging computational task, as there are 3×10^6 weights to be determined, not to mention a huge number of training samples to deal with. Most of the complexity of Word2Vec comes from tricks that are used to make it feasible to train such a large network with a massive amount of data.

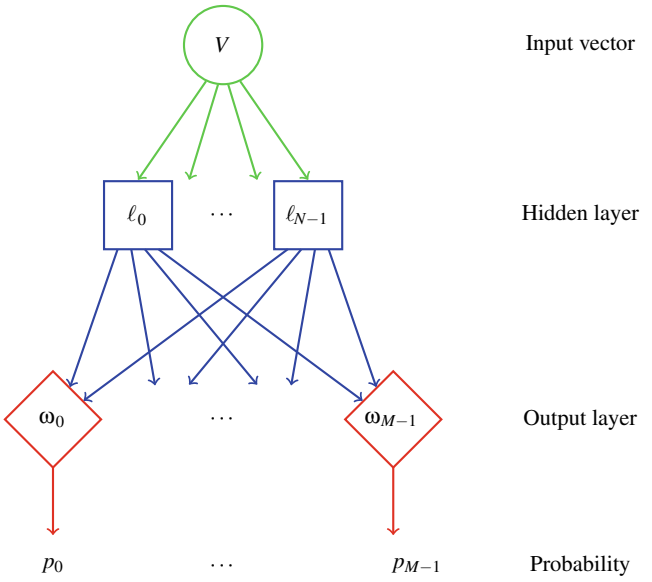


Fig. 29 Neural network for Word2Vec embeddings

One trick that is used to speed training in Word2Vec is the subsampling of frequent words. Common words such as "a" and "the" contribute little to the model, so these words can appear in training pairs at a much lower rate than they are present in the training text.

The most significant work-saving trick that is used in Word2Vec is so-called "negative sampling." When training a neural network, each training sample potentially affects all of the weights of the model. Instead of adjusting all of the weights, in Word2Vec, only a small number of "negative" samples have their weights modified per training sample. For example, suppose that the output vector of a training pair corresponds to word w_0. Then the "positive" weights are those of output node ω_0, and all of the corresponding weights are modified. In addition, a small subset of the $M - 1$ "negative" words (i.e., every word in the dataset except w_0) are selected and only the weights of the corresponding output nodes are modfied. The distribution used to select the negative subset is biased toward more frequent words.

A high-level discussion of Word2Vec can be found in [3], while a very nice and intuitive—yet reasonably detailed—introduction is given in [51]. The original paper describing Word2Vec is [53] and an immediate follow-up paper discusses a variety of improvements that mostly serve to make training practical for large datasets [54].

11.4 Word Embeddings in Malware Analysis

Word2Vec is fairly popular in the malware detection literature. For example, in [64] Word2Vec models based on machine code form the basis for a malware detection technique, while in [12], an Android malware detection scheme dubbed Droid-VecDeep uses Word2Vec results as features in deep belief networks [29]. The recent malware research in [9] considers multiple word embedding techniques (Word2Vec, HMM2Vec, and PCA2Vec) based on opcode sequences. Better results are obtained in most cases, as compared to using raw opcode sequences, which indicates that word embeddings are a useful form of feature engineering. The paper [36] considers Word2Vec and HMM2Vec embeddings for malware classification, with strong results obtained in many cases. In [62], word embeddings are used as part of a scheme that can successfully distinguish points in time where significant evolution has occurred within a malware family.

Word2Vec has proven surprisingly useful in a variety of security applications beyond the malware domain. Such applications range from network-based anomaly detection [4] to analyzing the evolution of cyberattacks [73].

12 Conclusion

In this chapter, we have provided details on a wide array of deep learning techniques that have proven useful in the field of malware analysis. We began with an introduction to the historical development of neural network-based techniques and related topics. This was followed by a discussion of several popular modern architectures. Specifically, we covered the following architectures: Multilayer perceptrons (MLP), convolutional neural networks (CNN), recurrent neural networks (RNN), long short-term memory (LSTM), gated recurrent units (GRU), residual networks (ResNet), generative adversarial networks (GAN), extreme learning machines (ELM), and Word2Vec. For each of these architectures, we cited representative examples of relevant malware-related research, and in most cases, we also mentioned other applications related to information security.

References

1. Annapurna, Annadatha, and Mark Stamp. 2018. Image spam analysis and detection. *Journal of Computer Virology and Hacking Techniques* 14 (1): 39–52.
2. Ben Athiwaratkun and Jack W. Stokes. 2017. Malware classification with LSTM and GRU language models and a character-level CNN. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/07/LstmGruCnnMalwareClassifier.pdf.
3. Banerjee, Suvro. 2018. Word2vec — A baby step in deep learning but a giant leap towards natural language processing. https://medium.com/explore-artificial-intelligence/word2vec-

a-baby-step-in-deep-learning-but-a-giant-leap-towards-natural-language-processing-40fe4e8602ba.
4. Barot, Ketul, Jialing Zhang, and Seung Woo Son. 2016. Using natural language processing models for understanding network anomalies. http://ieee-hpec.org/2016/techprog2016/index_htm_files/R-w2vec-final.pdf.
5. Basole, Samanvitha, Fabio Di Troia, and Mark Stamp. 2020. Multifamily malware models. *Journal of Computer Virology and Hacking Techniques* 16 (1): 79–92.
6. Bhodia, Niket, Pratikkumar Prajapati, Fabio Di Troia, and Mark Stamp. 2019. Transfer learning for image-based malware classification. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy*, ICISSP 2019, eds. Paolo Mori, Steven Furnell, and Olivier Camp, 719–726.
7. The Brown corpus of standard American English. http://www.cs.toronto.edu/~gpenn/csc401/a1res.html.
8. Cave, Robert L., and Lee P. Neuwirth. 1980. Hidden Markov models for English. In *Hidden Markov models for speech*, 16–56, IDA-CRD. New Jersey: Princeton. https://www.cs.sjsu.edu/~stamp/RUA/CaveNeuwirth/index.html.
9. Chandak, Aniket, Fabio Di Troia, and Mark Stamp. 2020. A comparison of word embedding techniques for malware classification. In *Malware analysis using artificial intelligence and deep learning*, eds. Stamp, Mark, Mamoun Alazab, and Andrii Shalaginov. Berlin: Springer.
10. Chavda, Aneri, Katerina Potika, Fabio Di Troia, and Mark Stamp. 2018. Support vector machines for image spam analysis. In *Proceedings of the 15th international joint conference on e-business and telecommunications*, ICETE 2018, eds. Callegari, Christian, Marten van Sinderen, Paulo Novais, Panagiotis G. Sarigiannidis, Sebastiano Battiato, Ángel Serrano Sánchez de León, Pascal Lorenz, and Mohammad S. Obaidat, 597–607.
11. Chen, Rui, Jing Yang, Rong-gui Hu, and Shu-guang Huang. 2013. A novel lstm-rnn decoding algorithm in CAPTCHA recognition. https://ieeexplore.ieee.org/document/6840561.
12. Chen, T., Q. Mao, M. Lv, H. Cheng, and Y. Li. 2019. Droidvecdeep: Android malware detection based on Word2Vec and deep belief network. *KSII Transactions on Internet and Information Systems* 13 (4): 2180–2197.
13. Cheng, Min, Qian Xu, Jianming Lv, Wenyin Liu, Qing Li, and Jianping Wang. 2016. MS-LSTM: A multi-scale LSTM model for BGP anomaly detection. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*, 1–6.
14. Cohen, Steven A., and Matthew W. Granade. 2018. Models will run the world. *Wall Street Journal*. https://www.wsj.com/articles/models-will-run-the-world-1534716720.
15. Cornelisse, Daphne. 2018. An intuitive guide to convolutional neural networks. https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050.
16. Deshpande, Adit. 2018. A beginner's guide to understanding convolutional neural networks. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/.
17. Extreme learning machine implementation in Python. https://github.com/dclambert/Python-ELM.
18. Fernández-Navarro, Francisco, César Hervás-Martinez, Javier Sanchez-Monedero, and Pedro Antonio Gutiérrez. 2011. MELM-GRBF: A modified version of the extreme learning machine for generalized radial basis function neural networks. *Neurocomputing* 74(16): 2502–2510.
19. Gasmi, Houssem, Jannik Laval, and Abdelaziz Bouras. 2019. Cold-start cybersecurity ontology population using information extraction with LSTM. In *2019 international conference on cyber security for emerging technologies*, CSET, 1–6.
20. Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. Cambridge: MIT Press. http://www.deeplearningbook.org.
21. Goodfellow, Ian J, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of the 27th international conference on neural information processing systems*, *NIPS'14*, vol. 2, 2672–2680.

22. Gormley, Matthew R. 2017. Neural networks and backpropagation. https://www.cs.cmu.edu/~mgormley/courses/10601-s17/slides/lecture20-backprop.pdf.
23. Greff, Klaus, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2017. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems* 28 (10): 2222–2232. https://arxiv.org/pdf/1503.04069.pdf.
24. Huang, Guang-Bin, Qin-Yu Zhu, and Chee-Kheong Siew. 2004. Extreme learning machine: A new learning scheme of feedforward neural networks. In *2004 IEEE international joint conference on neural networks*, vol. 2, 985–990.
25. Gupta, Arpit. 2018. Alexa blogs: How Alexa is learning to converse more naturally. https://developer.amazon.com/blogs/alexa/post/15bf7d2a-5e5c-4d43-90ae-c2596c9cc3a6/how-alexa-is-learning-to-converse-more-naturally.
26. Hardesty, Larry. 2017. Explained: Neural networks. http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.
27. He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. https://arxiv.org/pdf/1512.03385.pdf.
28. Hern, Alex. 2017. *The guardian*. Elon Musk says AI could lead to third world war. https://www.theguardian.com/technology/2017/sep/04/elon-musk-ai-third-world-war-vladimir-putin.
29. Hinton, Geoffrey. 2007. Deep belief nets. https://www.cs.toronto.edu/~hinton/nipstutorial/nipstut3.pdf.
30. Hochreite, Sepp and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9(8): 1735–1780. http://www.bioinf.jku.at/publications/older/2604.pdf.
31. Hu, Weiwei and Ying Tan. 2017. Black-box attacks against RNN based malware detection algorithms. https://arxiv.org/abs/1705.08131.
32. Hu, Weiwei and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on gan. https://arxiv.org/pdf/1702.05983.pdf.
33. Jahromi, Amir Namavar, Sattar Hashemi, Ali Dehghantanha, Kim-Kwang Raymond Choo, Hadis Karimipour, David Ellis Newton, and Reza M. Parizi. 2019. An improved two-hidden-layer extreme learning machine for malware hunting. *Computers and Security* 89.
34. Jain, Mugdha, William Andreopoulos, and Mark Stamp. Convolutional neural networks and extreme learning machines for malware classification. *Journal of Computer Virology and Hacking Techniques*.
35. Kaan, Can. 2018. Deep learning tutorial for beginners. https://www.kaggle.com/kanncaa1/deep-learning-tutorial-for-beginners.
36. Kale, Aparna Sunil, Fabio Di Troia, and Mark Stamp. 2020. Malware classification with hmm2vec and word2vec features. submitted for publication.
37. Kalfas, Ioannis. 2018. Modeling visual neurons with convolutional neural networks. https://towardsdatascience.com/modeling-visual-neurons-with-convolutional-neural-networks-e9c01ddfdfa7.
38. Karpathy, Andrej. 2018. Convolutional neural networks for visual recognition. http://cs231n.github.io/convolutional-networks/.
39. Khaitan, Pranav. 2016. Google AI blog: Chat smarter with Allo. https://ai.googleblog.com/2016/05/chat-smarter-with-allo.html.
40. Khan, Riaz Ullah, Xiaosong Zhang, and Rajesh Kumar. 2019. Analysis of resnet and googlenet models for malware detection. *Journal of Computer Virology and Hacking Techniques* 15 (1): 29–57.
41. Kim, Gyuwan, Hayoon Yi, Jangho Lee, Yunheung Paek, and Sungroh Yoon. 2016. LSTM-based system-call language modeling and robust ensemble method for designing host-based intrusion detection systems. https://arxiv.org/abs/1611.01726.
42. Kim, Jin-Young, Bu Seok-Jun, and Sung-Bae Cho. 2018. Zero-day malware detection using transferred generative adversarial networks based on deep autoencoders. *Information Sciences* 460–461: 83–102.
43. Kravchik, Moshe, and Asaf Shabtai. 2018. Detecting cyberattacks in industrial control systems using convolutional neural networks. https://arxiv.org/pdf/1806.08110.pdf.

44. Kurenkov, Andrey. 2015. A 'brief' history of neural nets and deep learning. http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/.
45. Levy, Omer, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics* 3: 211–225. https://levyomer.files.wordpress.com/2015/03/improving-distributional-similarity-tacl-2015.pdf.
46. Levy, Steven. 2016. The iBrain is here—and it's already inside your phone. *Wired*. https://www.wired.com/2016/08/an-exclusive-look-at-how-ai-and-machine-learning-work-at-apple/.
47. Li, Fei-Fei, Justin Johnson, and Serena Yeung. 2017. Lecture 10: Recurrent neural networks. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf.
48. Li, Fei-Fei, Justin Johnson, and Serena Yeung. 2017. Lecture 13: Generative models. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf.
49. Li, Shixuan, and Dongmei Zhao. 2019. A LSTM-based method for comprehension and evaluation of network security situation. In *2019 18th IEEE international conference on trust, security and privacy in computing and communications*, 723–728.
50. Lu, Renjie. 2019. Malware detection with lstm using opcode language. https://arxiv.org/abs/1906.04593.
51. McCormick, Chris. 2016. Word2vec tutorial — The skip-gram model. http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.
52. McCulloch , Warren S, and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5. https://pdfs.semanticscholar.org/5272/8a99829792c3272043842455f3a110e841b1.pdf.
53. Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. https://arxiv.org/abs/1301.3781.
54. Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.
55. Minsky, Marvin, and Seymour Papert. 1969. *Perceptrons: An introduction to computational geometry*. Cambridge: MIT Press.
56. Moody, Chris. Stop using word2vec. https://multithreaded.stitchfix.com/blog/2017/10/18/stop-using-word2vec/.
57. Moradi, Mehdi, and Mohammad Zulkernine. A neural network based system for intrusion detection and classification of attacks. https://pdfs.semanticscholar.org/cbf2/57a638aff38eae99bf88d8e22f150d9d8c47.pdf.
58. Narwekar, Abhishek, and Anusri Pampari. 2016. Recurrent neural network architectures. http://slazebni.cs.illinois.edu/spring17/lec20_rnn.pdf.
59. Neubig, Graham. 2018. NLP programming tutorial 8 — Recurrent neural nets. http://www.phontron.com/slides/nlp-programming-en-08-rnn.pdf.
60. Ng , Andrew Y, and Michael I. Jordan. 2001. On discriminative vs. generative classifiers: A comparison of logistic regression and naïve Bayes. In *Proceedings of the 14th international conference on neural information processing systems: natural and synthetic*, NIPS'01, 841–848.
61. Olah, Christopher. 2014. Understanding convolutions. http://colah.github.io/posts/2014-07-Understanding-Convolutions/.
62. Paul, Sunhera, Fabio Di, and Troia Mark Stamp. Word embedding techniques for malware evolution detection. submitted for publication.
63. Philipp, George, Dawn Song, and Jaime G. Carbonell. 2018. The exploding gradient problem demystified — Definition, prevalence, impact, origin, tradeoffs, and solutions. https://arxiv.org/pdf/1712.05577.pdf.
64. Popov, I. 2017. Malware detection using machine learning based on Word2Vec embeddings of machine code instructions. In *2017 Siberian symposium on data science and engineering*, SSDSE, 1–4.
65. Rabiner, Lawrence R. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77(2): 257–286. https://www.cs.sjsu.edu/~stamp/RUA/Rabiner.pdf.

66. Rezende, E, G. Ruppert, T. Carvalho, F. Ramos, and P. de Geus. 2017. Malicious software classification using transfer learning of resnet-50 deep neural network. In *16th IEEE international conference on machine learning and applications*, ICMLA 2017, 1011–1014.
67. Rigaki, Maria, and Sebastian Garcia. 2018. Bringing a GAN to a knife-fight: Adapting malware communication to avoid detection. https://mariarigaki.github.io/publication/gan-knife-fight/.
68. Rosenblatt, Frank. 1961. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. http://www.dtic.mil/dtic/tr/fulltext/u2/256582.pdf.
69. Ruderman, Avraham, Neil C. Rabinowitz, Ari S. Morcos, and Daniel Zoran. 2018. Pooling is neither necessary nor sufficient for appropriate deformation stability in CNNs. https://arxiv.org/abs/1804.04438.
70. Rumelhart, David, Geoffrey Hinton, and Ronald Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (9)
71. Shamshirband, Shahab, and Anthony T. Chronopoulos. 2019. A new malware detection system using a high performance-elm method. In *Proceedings of the 23rd international database applications & engineering symposium*, IDEAS'19, 33:1–33:10.
72. Sharmin, Tazmina, Fabio Di Troia, Katerina Potika, and Mark Stamp. 2020. Convolutional neural networks for image spam detection. *Information Security Journal: A Global Perspective* 29 (3): 103–117.
73. Shen, Yun, and Gianluca Stringhini. 2019. Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks. https://seclab.bu.edu/people/gianluca/papers/attack2vec-usenix2019.pdf.
74. Singh, Tanuvir, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. 2016. Support vector machines and malware detection. *Journal of Computer Virology and Hacking Techniques* 12 (4): 203–212.
75. Springenber, Jost Tobias, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. https://arxiv.org/abs/1412.6806.
76. Spruston, Nelson. 2019. Pyramidal neurons: Dendritic structure and synaptic integration. *Nature Reviews Neuroscience* 9: 206–221. https://www.nature.com/articles/nrn2286.
77. Stamp, Mark. 2004. A revealing introduction to hidden Markov models. https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf.
78. Stamp, Mark. 2018. A survey of machine learning algorithms and their application in information security. In *Guide to vulnerability analysis for computer networks and systems: an artificial intelligence approach*, eds. Parkinson, Simon, Andrew Crampton, and Richard Hill, chapter 2, 33–55. Berlin: Springer.
79. Stamp, Mark. 2019. Alphabet soup of deep learning topics. https://www.cs.sjsu.edu/~stamp/RUA/alpha.pdf.
80. Stamp, Mark. 2019. Deep thoughts on deep learning. https://www.cs.sjsu.edu/~stamp/RUA/ann.pdf.
81. Veit, Andreas, Michael Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. https://arxiv.org/pdf/1605.06431.pdf.
82. Wallis, Charles. 2017. History of the perceptron. https://web.csulb.edu/~cwallis/artificialn/History.htm.
83. Wu, Peilun, Hui Guo, and Nour Moustafa. 2020. Pelican: A deep residual network for network intrusion detection. https://arxiv.org/pdf/2001.08523.pdf.
84. Wu, Yonghui, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. https://arxiv.org/abs/1609.08144.
85. Xu, Ke, Yingjiu Li, Robert H. Deng, and Kai Chen. 2018. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European symposium on security and privacy*, Euro SP, 473–487.

86. Xue, Di, Jingmei Li, Tu Lv, Weifei Wu, and JiaXiang Wang. 2019. Malware classification using probability scoring and machine learning. *IEEE Access*, 91641–91656.
87. Yagcioglu, Semih, Mehmet Saygin Seyfioglu, Begum Citamak, Batuhan Bardak, Seren Guldamlasioglu, Azmi Yuksel, and Emin Islam Tatli. 2019. Detecting cybersecurity events from noisy short text. https://arxiv.org/abs/1904.05054.
88. Yajamanam, Sravani, Vikash Raja Samuel Selvin, Fabio Di Troia, and Mark Stamp. 2018. Deep learning versus gist descriptors for image-based malware classification. In *Proceedings of the 4th international conference on information systems security and privacy*, ICISSP 2018, eds. Mori, Paolo, Steven Furnell, and Olivier Camp, 553–561.
89. Zeiler, Matthew D, and Rob Fergus. 2014. Visualizing and understanding convolutional networks. https://cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf.
90. Zhang, Wei, Huan Ren, Qingshan Jiang, and Kai Zhang. 2015. Exploring feature extraction and ELM in malware detection for Android devices. *Advances in Neural Networks, ISNN*, eds. Hu, Xiaolin, Yousheng Xia, Yunong Zhang, and Dongbin Zhao, 489–498.