SOPHOS
Security made simple.

Machine Learning: How to Build a Better Threat Detection Model

By Madeline Schiappa

At Sophos, we're focused on protecting our customers from threats from every possible attack vector. And here in the Data Science Group, we're challenged every day to come up with new and better techniques to address these cyber threats in a scalable way that not only improves protection, but changes the paradigm of how emerging threats are addressed. That's why we're focusing on new deep learning and machine learning methods to be leveraged across our entire portfolio.

One of our first challenges is supplementing reactive, human-based malware research with predictive machine learning models. This challenge is very unique, and can be an afterthought in traditional machine learning cybersecurity literature.

In this article, we describe the process we use to develop our models. To help explain the concepts, we'll work through the development and evaluation of a toy model meant to solve the very real problem of detecting malicious URLs.

Detecting Malicious URLs the Traditional Way

Let's start with how the problem of malicious URL detection can be traditionally solved using signatures, and then take a closer look at how we would design a detection model.

Say we have reports of the following URLs:

Malicious	Clean
\facebook.com-me12.xy\r\l\c580f098f8cfc54cd872a35192a82ac3\?entrypop=1	graph.facebook.com/10153752105048199/picture
\facebook.com.piihs.edu.bd\	2cv.facebook.com/
\login.facebook.whats-gratis.com\3849485691b6ff4908136fdcb303ff9f82d07ab4	connect.facebook.net/signals/config/1659451214312211
\login.facebook.whats-gratis.com\SPUkZ\VlQLZ\VPMUZ\	connect.facebook.net/sdk.js
\www.facebook.com.vnomera.com\widgets\like.php	www.intern.facebook.com/

Blacklisting

A traditional protection method would be to add the malicious URLs to a blacklist that is then either pushed out to customers directly or updated in a cloud-based blacklist service leveraged by an endpoint product.

The problem with these solutions is that the sheer daily volume of malicious URLs found on the internet means that updates can grow relatively large in size, which naturally leads to decreased performance on end users' machines due to increased disk and memory usage. Furthermore, since the internet is used to either push updates to customers or pull updates from cloud services, if connections get interrupted or updates don't complete correctly, customers can remain unprotected from URL updates. Additionally, in the instance of cloud-based lookups, round-trip latency delays can negatively impact the user experience. And perhaps the biggest issue with this method is that it's reactive: malicious URLs must be detected and protections published prior to users navigating to them.

Regular Expressions (RegEx) and Signatures

Another traditional method is to create regex-based signatures meant to capture malicious URLs and their variants. Similar to blacklisting, after signatures are created, they are delivered to customers either via an update or pushed from the cloud, meaning the same issues of connectivity and memory usage apply.

The main concern, however, is whether we base the regex match on the domain itself or include what comes after or before the domain as well. In the example URLs, we use several Facebook links as the target URLs being exploited. These examples demonstrate the importance of analysis, because if we were to simply create a signature that blocks traffic based on "facebook.com" in general, we would wholesale block a commonly used, popular, clean site. Facebook – and those who visit it often – would be very unhappy.

However, we still want to protect customers from malicious content that may have been attached to the Facebook domain and related keywords. For example, we could write a signature that uses regex to block URLs that match "facebook.com" but that are followed by a period with more text after the initial ".com" portion of the URL. This would block two of our sample URLs out of the five.

We could further finesse this signature to block URLs containing any periods, hyphens, or text that directly followed "facebook.com" without the presence of a slash. The regex "/^facebook\.com[\-\.\w\/\?\=]+$/" would block three out of five of our sample URLs:

1. \facebook.com.piihs.edu.bd\
2. \facebook.com-me12.xy\r\l\c580f098f8cfc54cd872a35192a82ac3\?entrypop=1
3. \www.facebook.com.vnomera.com\widgets\like.php'.

Because it only blocks three of the five URLs, an additional signature would be needed to capture the remaining two. These signatures each take about five minutes to write, meaning that two signatures will require an initial investment of 10 minutes just to block five URLs. When there are thousands of malicious URLs, this time adds up quickly. We also need to test the signatures to ensure they don't block the clean URLs, which can take another 5 minutes each.

As you can see, we're now up to 20 minutes. And this doesn't include the time it takes to find and validate which URLs are clean and which are malicious. Each time we receive a new set of malicious and clean URLs, our human analysts have to undergo the process all over again.

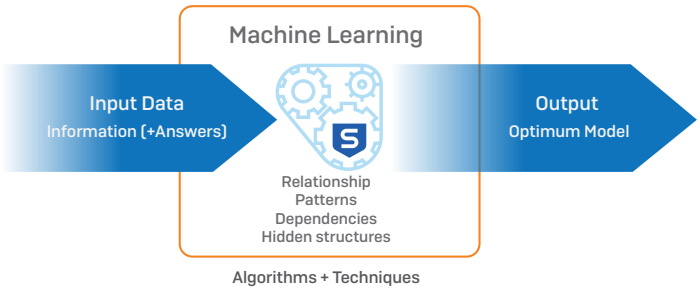
In summary, with this method, human analysts are constantly analyzing URLs, creating signatures, and pushing out updates. This causes several areas of concern:

1. This is a reactive method, so some customers may visit malicious sites before we know about them. Additionally, they may not be protected from zero-day malware.
2. An individual signature can only match on so many variants of a domain, resulting in the need for many, many signatures to cover only a portion of malicious URLs.
3. Because updates are pushed through an internet connection, there's always a risk of an interruption to an update, meaning customers may be unprotected from the latest malicious content. These updates also consume a lot of memory on the endpoint.
4. The manual generation and subsequent maintenance of these signatures is not only slow, but requires a large investment of time and resources.

A Brief Introduction to Machine Learning

Machine learning "learns" by using mathematical models instead of being explicitly programmed to address the particularities of a specific problem. Using large amounts of data, we generate a general model that is able to accurately describe the data it's ingesting. However, since we're dealing with general models in order to try to explain specific phenomena, we never know if our machine learning model has learned to predict properly. As such, any model that we develop is always coupled with a rigorous set of evaluations.

Here at Sophos, we focus specifically on deep learning, which is a kind of machine learning that most similarly mimics the human brain. Deep learning involves many layers of neurons to form an artificial neural network. Both a brain-based neural network and an artificial neural network ingest some sort of input, manipulate the input in some way, and then output information to other neurons. The major difference is that the human brain contains approximately 100 billion neurons, while an artificial neural network contains a miniscule fraction of that. In order to develop a meaningful deep learning model, we need to feed it large amounts of data, translate the date into a language that the model can understand, building the underlying architecture to support the model, and then finally train, test, and evaluate the model.



In our malicious URLs example, we can leverage large sets of data to recognize characteristics of benign and malicious URLs automatically. Eventually, our model will be able to predict the likelihood that a given URL is malicious without storing signatures or blacklists on the local machine. We're left with a generalized model that covers the entire distribution of data, whereas signatures can only detect small subsets of samples.

With our research, we are able to automate detection processes and push updates less frequently. Instead of analyzing a suspicious URL against many signatures for a possible match, it can be passed through our URL model and assigned a score based upon how malicious it appears. If the score is above a certain threshold, the URL will be blocked.

Customer machines don't need to be connected to the internet to receive updates every day in order to be protected. With deep learning, updates are just newly trained models based on the same feature engineering techniques; therefore, we can continuously improve the architecture of our model without redesigning its features. Features are extracted continuously and easily without requiring changes to our collection method, and changes to the model itself are largely unnecessary. We simply retrain the model so it can predict what's next in the current landscape.

Feature Engineering in Machine Learning

Before creating a machine learning model, it's important to prepare our data. Preparing the data requires translating it into a language our model can understand. This is referred to as feature engineering.

Artificial neural network models intake data as a vector of information, so simply feeding the model a URL – which is not in the language of a vector – means that the model can't process it without some manipulation. There are countless ways that samples can be translated into features, though it takes some domain knowledge to do so. Using the URL example again, one way to translate a URL into a usable language is through a combination of ngramming and hashing. Ngrams are a popular method in DNA sequencing research. For example, the results of a three-gram ngram for the URL "https://sophos.com/company/careers.aspx" would be:

['htt', 'ttp', 'tps', 'ps:', 's:/', '://', '//s', '/so', 'sop', 'oph', 'pho', 'hos', 'os.', 's.c', '.co', 'com', 'om/', 'm/c', '/co', 'com', 'omp', 'mpa', 'pan', 'any', 'ny/', 'y/c', '/ca', 'car', 'are', 'ree', 'eer', 'ers', 'rs.', 's.a', '.as', 'asp', 'spx']

Once the ngrams are calculated, we need to translate them into a numerical representation. This can be done through a hashing mechanism. We will create an n-length long vector – say 1000 – and hash each ngram using a hashing algorithm. The resulting number from the hash of a particular ngram will be the index of which we will add 1. For example, if the first ngram 'htt' results in a hash of three and our vector is five units long, the result would be [0, 0, 1, 0, 0]. We continue this process for every ngram and for every URL until we have the list of URLs completely transformed into individual n-length vectors. When using this method for our toy model, these vectors will be 1,000 units long.

Artificial Neural Networks

Deep learning typically refers to three major components that, when combined together, allow for the creation of very powerful predictive models:

1. A connected graph of layers wherein each layer takes input from a parent layer, mixes the data together in some predefined way, and outputs it to the next layer in the graph
2. A loss function that measures how accurate the model makes its predictions
3. An algorithm that optimizes the loss function and trained dataset

Layers

Layers are made of interconnected nodes, or neurons. Each layer is some differentiable function that takes in a set of input weights, does some basic manipulation, and outputs the result as a set of output weights. Layers can be split into two categories: (1) layers that mix together input weights or (2) activation functions that independently act upon each input weight.

A layer that mixes input weights together is known as a dense layer. A dense layer exists when all the neurons in a particular layer are connected to all those in the next layer. For example, one neuron in this layer could mix together inputs [1, 2, 3] with weights [.5, .5, 1] to result in an output of [.5, 1, 3] after the inputs and weights are multiplied. In Figure 1, the weights input into a neuron are displayed next to the letter W alongside the arrows pointing to the neurons.

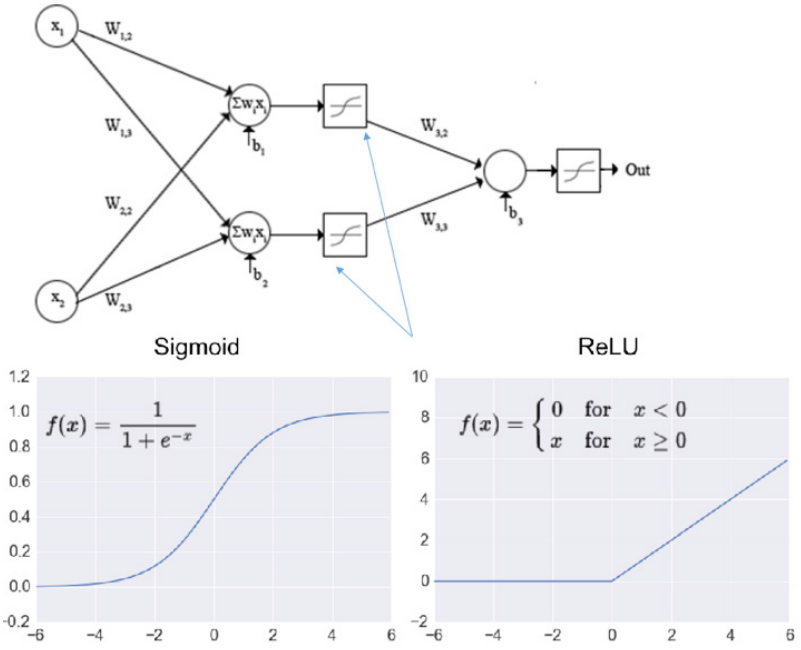
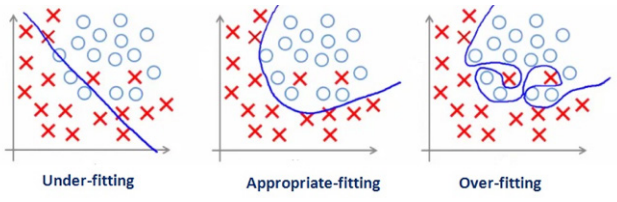


Figure 1: Sigmoid and ReLU are both commonly used activation functions

The next layer is known as the activation layer. The results from the previous layer are fed into the activation function associated with this layer to provide an output. The different activation functions available are softmax, ReLU, tanh, ELU, sigmoid, linear, softplus, softsign, and hard sigmoid. For hidden layers, sigmoid and ReLU are both commonly used activation functions. Sigmoid ranges from 0 to 1, while ReLU ranges from 0 to infinity. Deep learning commonly uses ReLU because it handles certain constraints better than sigmoid.



Simply combining layers that we described above typically results in overfitting. Overfitting occurs when our model learns only the training data but does not perform well on any new data. This is why almost every deep neural network is regularized in some way. We can regularize the network by either directly regularizing the weights inside a layer (for example, L1 or L2 regularization), or we can put regularization layers in between standard layers.

Two commonly used regularization layers are dropout layer and batch normalization layer. Dropout layer is a regularization used to reduce overfitting the model against the training set, and serves to help the model improve its prediction generalization when working with new datasets.

Dropout works by randomly dropping a designated percentage of weights to zero, which helps neurons learn different things from the data. By combining the neurons, the model produces a stronger classifier and ensures the overall network will not depend on one neuron alone.

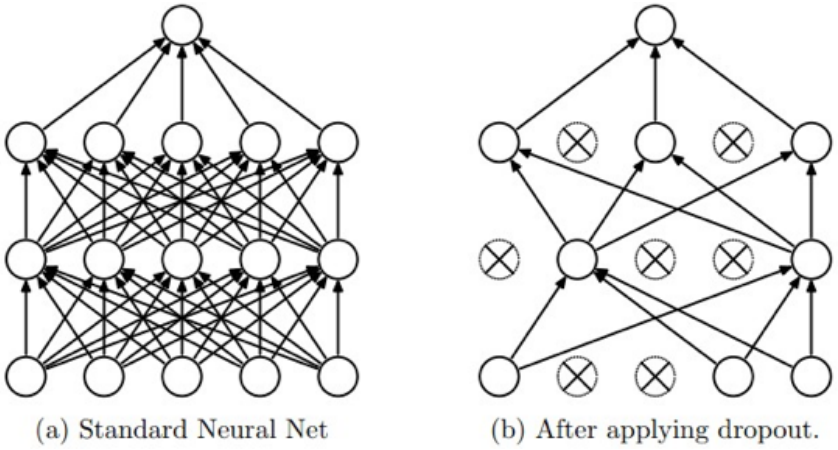


Figure 2: The impact of dropouts on neural networks

Batch normalization regulates batches of input before sending them to the next layer, resulting in each batch having a mean of zero and a standard deviation of one. This can accelerate learning and improve accuracy by removing certain outliers. Read more on batch normalization.

Loss Function

Once we lay out a model graph, we need to train the model to accurately classify the results. In our example here, we need to train our model to properly distinguish between good and bad URLs. The first thing we need is a way to measure how successful our model is during each step of the training process. This measurement, which needs to be differentiable, is referred to as a loss function. Various loss functions can be used for the same model, and each can potentially yield somewhat different results. For classification tasks, such as URL detection, the most common loss function used is cross-entropy.

Cross-entropy is used to quantify the difference, or loss, between the distribution of a model's predictions and the actual label's distribution. We are measuring how far away the model is from the optimal solution: where the prediction distribution and the actual distribution match.

When we use a sigmoid output as our final layer, we get an output of two probabilities for each URL: the probability that the URL is benign and the probability that the URL is malicious. Let's assume the threshold in this scenario is 0.5, meaning a probability greater than or equal to 0.5 is malicious and anything less is benign. We can then calculate the cross-entropy loss for each URL as depicted in the example below:

Probability	Prediction	Actual	Correct	Cross-entropy Loss
[0.3 0.7]	Malicious	[0 1] (malicious)	True	-(log(0.3)*0 + log(0.7)*1) = - log(0.7)=0.36
[0.6 0.4]	Benign	[1 0] (benign)	True	-(log(0.6)*1 + log(0.4)*0) = - log(0.6)=0.51
[0.2 0.8]	Malicious	[1 0] (benign)	False	-(log(0.2)*1 + log(0.8)*0) = - log(0.2)=1.6

What the model uses as the cross-entropy error is the average of all training samples. In this case, the average cross-entropy is: -(log(0.7) + log(0.6) + log(0.2))/3 = 0.83

Our goal is to minimize the average cross-entropy loss to improve the trustworthiness of our model.

Optimization

Optimization is the process of adjusting model weights in a way that minimizes the average loss over all the training samples. Imagine weights on horizontal axes and loss on a vertical axis, and for simplicity, the loss function looks like a parabolic bowl. The goal is to find the weights at the bottom of the bowl as depicted in the far-right image of Figure 3.

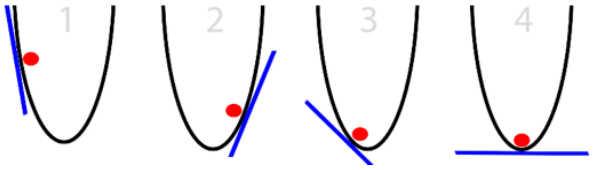
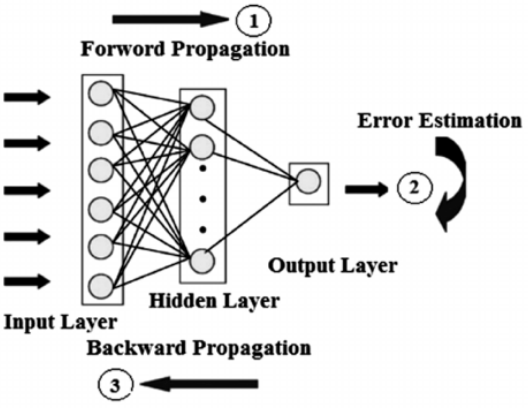


Figure 3

This method is called Stochastic Gradient Descent and is a process that updates weights through the gradient of the loss. The method by which you calculate the gradient of our loss function is called backpropagation.

This can be described in three steps:

1. Feed the model input and measure the error of the output using the loss function.
2. Update weights using the gradients; in other words, adjust them in a way that reduces the error.
3. Repeat this process for all training samples until the weights are no longer changing.



The mathematics behind this process are beyond the scope of this article, so we will not go into further detail. However, additional resources on the topic can be found here:

- https://www.youtube.com/watch?v=qtyiJTMcxao
- https://goo.gl/B4DM98
- https://goo.gl/vUokCZ

Optimizing the model requires feeding the model batches of data, and running over that data a certain number of times – also known as an epoch. Feeding the model is done in batches because the size of our data prevents it from being exposed to the algorithm computationally at one given time. The higher the batch size, the more memory needed. A single epoch means the algorithm has seen every input once. If epoch is set to 10, for example, the model will see each input 10 times. Batch size and number of epochs are two parameters that are decided before the fitting of the model begins. Once we have trained and optimized the model, we must then evaluate its performance to determine if it is ready for deployment to customers.

Evaluating the Performance

When a model predicts a URL as malicious, there is always a chance that the model is incorrect. Conversely, there is also a chance that the model predicts a URL as benign when it is actually malicious. Knowing how much to trust a model's decision is an important aspect of evaluating its performance.

When a URL is predicted to be malicious but is actually benign, the event is considered a false positive (FP). When a URL is predicted to be benign but is actually malicious, the event is considered a false negative (FN). Correctly classified malicious URLs are true positives (TP) and correctly classified benign URLs are true negatives (TN). These four categories are combined to create metrics that help evaluate our models.

Precision is one of the measures that gives us an idea about how trustworthy the model is. Precision is calculated using the following formula:

$$Precision = \frac{TP}{TP + FP}$$

Recall is a metric used to understand how many bad URLs the model missed, which gives a better picture of how well the model detects bad URLs. Recall is also known as the true positive rate (TPR). TPR is an indicator of all the bad URLs the model has seen and how many the model correctly labeled as bad. Recall is calculated using the following formula:

$$Recall = \frac{TP}{TP+FN}$$

Before deploying a model, a decision threshold is set. If the probability output from the model for the URL is greater or equal to the threshold, the URL is predicted malicious; if it is less than the threshold, it is predicted benign. We decide the threshold based on the desired false positive rate (FPR) that results when applied to the test dataset. The false positive rate is the rate at which the model will detect a URL that is actually benign. When we change that threshold, precision and recall will change as well because the count of TP, TN, FP and FN will change.

This is because URLs that were above the threshold may now be below, and vice versa. Generally, when recall increases, precision decreases. If the trustworthiness of the model is more highly valued, the threshold should result in higher precision; if the priority is catching bad URLs rather than trustworthiness of the prediction, then we choose a threshold that results in higher recall.

Another metric we most often use to evaluate our classification models is the receiver operating characteristic (ROC) curve. This measure is made up of TPR and FPR. This ratio indicates how many bad URLs the model has detected out of all the URLs it has seen. To calculate the TPR, use the recall formula mentioned earlier. The FPR is a ratio that describes the number of URLs our model detected as bad but that were actually good.

The FPR is calculated using the following formula:

$$FPR = \frac{FP}{FP+TN}.$$

The ROC curve uses TPR and FPR and plots them against each other using different thresholds, with the FPR on the x-axis and the TPR on the y-axis. A good ROC curve will start high in the top-left corner and continue growing higher as the FPR increases. There are two more metrics we can gather from the ROC curve: the optimal threshold for our model and the area under the ROC curve (AUC). The greater the AUC, the better. Therefore, a desirable model is one that has a very high AUC.

When should we use a ROC curve and when should we use precision and recall? That decision depends on the data and what we're looking to model. If our positive class – the item we're looking to detect – is small in comparison to the negative class, then precision and recall is a better tool. If our positive and negative classes are balanced, then ROC curve is a better choice. In this case, we train the model on balanced classes. The number of bad URLs and good URLs are similar, so the ROC curve would be a better measure.

Putting It All Together

Here we will walk through the process of:

1. Preparing the data: modifying input and feature engineering
2. Building the model
3. Training and validating the model
4. Evaluating the model

Preparing the Data

There are many tools that can be used for engineering features. The primary ones used are from a popular Python package called Scikit-learn (sklearn), which has several algorithms under the feature extraction section that can hash input into vectors. In the toy example, however, we use packages NLTK (a natural language toolkit) and mmh3 (a MurmurHash Python package) to encode each URL into a 1,000-length numerical vector.

Below is our hashing function for URLs:

```
def eng_hash(data, vdim=1000):
    final = []
    for url in data:
        v = [0] * vdim
        new = list(ngrams(url, 3))
        for i in new:
            new_ = ''.join(i)
            idx = mmh3.hash(new_) % vdim
            v[idx] += 1
        final.append([np.array(v)])
return final
```

Now that we have our feature vectors in a language our neural network can understand, we have to split the data into a training set and a testing set. The reason we split the data is because training a model to accurately explain training data is of little practical use, since what we're interested in is how a model performs on previously unseen data. We need to test the performance of the model on that unseen data, so we need a test set to evaluate that performance.

There are several ways to make this split, one being a random split. Sklearn has packages to assist in splitting the data as well. However, when concerning malware, we use a time split, which allows us to improve zero-day malware detection. The training set would be files first seen in a time period before the test set. The test set would be samples that the model has never seen before, playing the role as zero-day malware.

The model is optimized on good performance in predicting the test set, so the model ideally will generalize well to real zero-day malware. More on this method can be found in "Improving Zero-Day Malware Testing Methodology Using Statistically Significant Time-Lagged Test Samples" by Sophos data scientists Joshua Saxe and Konstantin Berlin.

An example of this in action is depicted below:

```
p_cut = 70.0
    percentile = np.min((np.percentile(first_seen[y_label==0],
p_cut), np.percentile(first_seen[y_label==1], p_cut)))

    train = []
    test = []
    for i, v in enumerate(first_seen):
        # train only on benign data
        if v < percentile and y_label[0][i] >= 0:
            train.append(i)
        elif v >= percentile and y_label[0][i] >= 0:
            test.append(i)

    cv = [[np.array(train), np.array(test)]]
```

Building the Model

We have our data prepared, so now we will build the model. The major tool our team uses for developing deep neural networks is Keras. Keras uses Google's Tensorflow (TF) as the primary backend engine and can work along with TF libraries when building and training models. Keras abstracts TF building blocks, allowing us to build a model layer by layer.

The first step to creating a model using Keras is to initialize the model:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization,
Activation

model = Sequential()
```

At each layer, we determine which function to use and how many neurons it contains. A common layer used is the fully connected layer, called the dense layer. Dense layer means all the neurons in a layer are connected to all those in the next layer.

In Keras, the first layer must also have the input dimensions defined. For example, if we're feeding the model a vector of length 1,000 then the input dimension would be equal to 1,000. Regularization methods are added by using Keras layers as well – therefore activation, dropout, and normalization of our batches are added sequentially.

In our example, we will randomly drop 15% of the weights and use the ReLU activation function. An example of one hidden layer is below:

```
model.add(Dense(128, input_dim=1000))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(.15))
```

Deep neural networks use many of these hidden layers. After the hidden layers are written, the loss function, optimization function, and any metrics we want to collect are compiled with the model.

In our example, we use cross-entropy loss and stochastic gradient descent for optimization:

```
model.compile(loss='binary_crossentropy',
                  optimizer='SGD',
                  metrics=['accuracy'])
```

Training and Evaluating

Now we have our training data set, testing data set, and our compiled model. We're ready to train. In our example, we choose a batch size of 128 with 20 epochs. This means that samples are fed into the model in batches of 128 vectors and we will pass through the entire training set 20 times. We feed the model the training URL vectors and we compare the model's output to the actual labels – 1 for malicious and 0 for benign depicted as y train – to calculate prediction error:

```
def train_model(X_train, y_train, model):
    log.info("Beginning training model")
    loss = LossHistory()
    model.fit(X_train, y_train,
              epochs=20,
              batch_size=128, verbose=1, callbacks=[loss])
    return model, loss
```

The model learns through backpropagation, minimizing the cross-entropy error, to find the optimal weights. After completion of the training, the model is then tested on the test set, and metrics are collected to evaluate the model and to help investigate any possible improvements.

The first metrics we will use are precision and recall. Precision tells us how trustworthy the model is and recall tells us how many bad URLs we detected:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN}$$

Here are some results from our deep layered model using a time split with a threshold that results in a false positive rate of 1e-4, which is equivalent to a false positive occurring .01% of the time and true positive rate of approximately 29%:

	Predicted Labels From our Model		
Actual Labels	*FPR: 1e-4	Benign	Malicious
	Benign	595,305	59
	Malicious	461,482	186,557

The precision of our test model is URLs predicted malicious that are indeed malicious divided by the total URLs classified as bad. In the case of the above table, TP=186,557 and FP=59. Therefore to calculate the precision, we take 186,557/(186,557+59) = 0.99968, indicating the model is trustworthy about 99.9% of the time. To calculate recall, we use TP=186,557 and FN=461,482 to get 186,557/(186,557+461,482) = 0.2878, indicating the model only caught 28.9% of the bad URLs.

When we move the threshold to a false positive rate of 1e-3, which is equivalent to a false positive occurring .1% of the time, the results will change – the true positive rate changing to approximately 63%. The counts are depicted in the below table:

	Predicted Labels From our Model		
Actual Labels	*FPR: 1e-3	Benign	Malicious
	Benign	594,769	595
	Malicious	242,064	405,975

In this case, the precision is: 405,975/(405,975+595) = 0.9985, indicating the model is trustworthy about 99.8% of the time. Recall is 405,975/(405,975+242,064) = 0.6265, indicating the model only caught 62.7% of bad URLs.

This example shows how changing the threshold can result in significant changes to recall and precision. When the false positive rate was fixed at 1e-4, we could trust our model 99.9% of the time but the detection rate was 28.9%. When the false positive rate was fixed at 1e-3, we could trust our model 99.8% of the time and the detection rate was 62.7%! These two thresholds are very close to each other. If we picked ones further apart, we would see much larger differences.

When evaluating our models, we generally focus on the ROC and AUC as the evaluation metrics. These calculations are simple to compute but are tedious and are prone to errors. It's more efficient and reliable to utilize the Scikit-learn Python library to calculate these metrics for us. The Sklearn library provides a suite of utility tools and helper function to get the precision, recall, and the roc curve. These utility functions can be found under the sklearn.metrics package with function names roc_curve, precision_score, and recall_score:

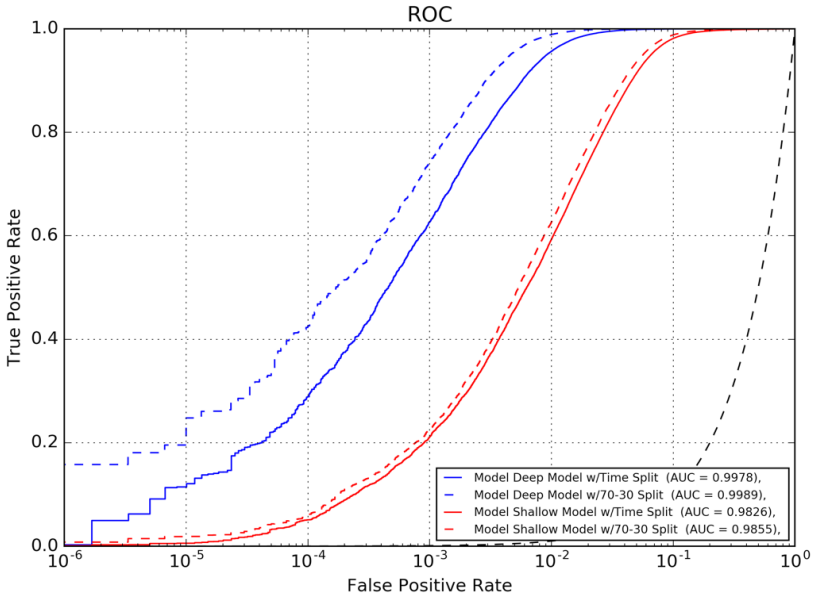
```
    from sklearn.metrics import roc_curve, auc

preds =  model.predict(X_test, batch_size=64)
preds = preds.astype(np.float)

#get roc curve using sklearn
results = {}
fpr, tpr, thresh = roc_curve(y_test, preds[:,1], 1.0)
curr_auc = auc(fpr, tpr)
```

With these metrics calculated, we can now generate a plot using matplotlib for Python to visualize the results. This is also helpful in comparing multiple models. In this example, we are going to compare a shallow model that has one sigmoid output layer and a deeper model with three hidden layers and a sigmoid output layer. We will also compare results for a time split of the training and testing data, and a random split.

The results of our example URL model are shown in the ROC curve below, showing a better performance from the deep learning model relative to the shallow model. Using plots like this allows us to compare model architectures and threshold parameters in order to deploy the best model to our customers. The plot is zoomed in to focus on the FPRs that we are interested in for deployment. From the plot, you can see that the deep model performs much better than the shallow model with an AUC around .99, while the shallow model has an AUC around .98. You can also see the tradeoff between FPR and TPR. Making decisions about which threshold to deploy depends on this tradeoff. The URL model deployed to our customers will perform at much higher rates of detection than depicted below.



Try It Yourself

The example that we just walked you through here is available on GitHub.

Summary

We've discussed here the challenges of detecting malicious content and how machine learning is a supplementary tool that can help address these challenges. As opposed to more traditional methods of blacklisting and handwritten signatures to detect malware, machine learning addresses these challenges by delivering a low-maintenance, automated system and by capturing a larger distribution of malware.

Specifically, we introduced the concept of deep learning and how we can quickly build, train, and evaluate models end-to-end using recently developed packages like Keras, Tensorflow, and Scikit-learn. We demonstrated this end-to-end process through our example URL model and how complex, deep models allow us to improve detection compared to more traditional, shallow models.

Sophos' investment in deep learning research is a commitment to protect our customers using the latest science advancements, recognizing that machine learning is the future of the industry. When our deep learning expertise is combined with our deep history in endpoint protection, it uniquely positions Sophos as the market leader in next-generation detection tools. This serves our ultimate goal: keeping customers protected from evolving threats.

Learn More

Read our threat research and data science technical papers.

United Kingdom and Worldwide Sales
Tel: +44 (0)8447 671131
Email: sales@sophos.com

North American Sales
Toll Free: 1-866-866-2802
Email: nasales@sophos.com

Australia and New Zealand Sales
Tel: +61 2 9409 9100
Email: sales@sophos.com.au

Asia Sales
Tel: +65 62244168
Email: salesasia@sophos.com

Oxford, UK

2017-08-01 (MP)

SOPHOS