9
Bypassing Machine Learning Malware Detectors

In the previous chapter, you learned that you can break into machine learning models and make them perform malicious activities by using adversarial machine learning techniques. In this chapter, we are going to explore further techniques, like how to fool artificial neural networks and deep learning networks. We are going to look at anti-malware system evasion as a case study.

In this chapter, we will cover the following:

- Adversarial deep learning
- How to bypass next generation malware detectors with generative adversarial networks
- Bypassing machine learning with reinforcement learning

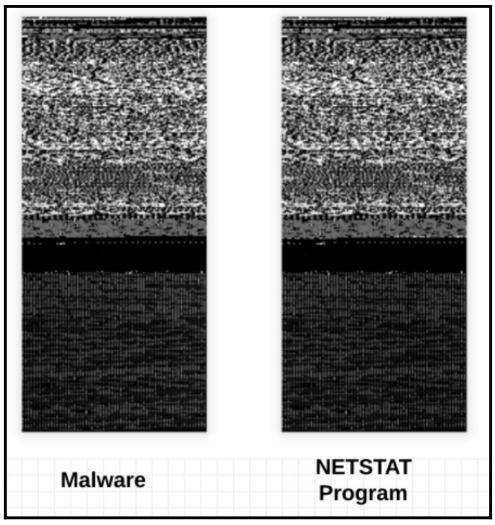
Technical requirements

You can find the code files for this chapter at `https://github.com/PacktPublishing/Mastering-Machine-Learning-for-Penetration-Testing/tree/master/Chapter09`.

Adversarial deep learning

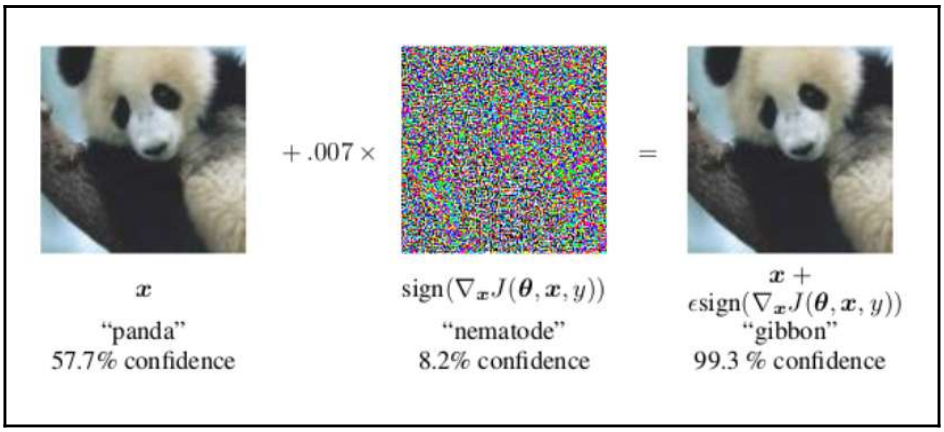
Information security professionals are doing their best to come up with novel techniques to detect malware and malicious software. One of the trending techniques is using the power of machine learning algorithms to detect malware. On the other hand, attackers and cyber criminals are also coming up with new approaches to bypass next-generation systems. In the previous chapter, we looked at how to attack machine learning models and how to bypass intrusion detection systems.

Malware developers use many techniques to bypass machine learning malware detectors. Previously, we explored an approach to build malware classifiers by training the system with grayscale image vectors. In a demonstration done by the **Search And RetrieVAl of Malware** (**SARVAM**) research unit, at the Vision Research Lab, UCSB, the researchers illustrated that, by changing a few bytes, a model can classify a malware as a goodware. This technique can be performed by attackers to bypass malware classifiers, through changing a few bytes and pixels. In the demonstration, the researchers used a variant of the NETSTAT program, which is a command-line network utility tool that displays network connections. In the following image, the left-hand side is a representation of the `NETSTAT.EXE` malware, and the second is detected as a goodware. As you can see, the difference between the two programs is unnoticeable (88 bytes out of 36,864 bytes: 0.78%), after converting the two types of files into grayscale images and checking the differences between the two of them:



This technique is just the beginning; in this chapter, we are going to dive deep into how we can trick them (the machine learning model, in our case the malware classifier) into performing malicious activities.

The previous chapter was an overview of adversarial machine learning. We learned how machine learning can be bypassed by attackers. In this chapter, we are going to go deeper, discovering how to bypass malware machine learning based detectors; before that, we are going to learn how to fool artificial neural networks and avoid deep learning networks with Python, open source libraries, and open source projects. Neural networks can be tricked by **adversarial samples**. Adversarial samples are used as inputs to the neural network, to influence the learning outcome. A pioneering research project, called *Explaining and Harnessing Adversarial Networks,* conducted by Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy (at Google), showed that a small amount of carefully constructed noise can fool the neural network into thinking that the entered image is an image of a gibbon and not a panda, with 99.3% confidence. The neural network originally thought that the provided image was a panda, with 57.7% confidence, which is true; but it is not the case in the second example, after fooling the network:



Many electronic devices and systems rely on deep learning as a protection mechanism, including face recognition; imagine what attackers can do to attack them and gain unauthorized access to critical systems.

Now, let's try to fool a neural network. We are going to fool a handwritten digit detector system by using the famous MNIST dataset. In `Chapter 4`, *Malware Detection with Deep Learning*, we learned how to build one. For the demonstration, we are going to fool a pretrained neural network by Michael Nielsen. He used 50,000 training images and 10,000 test images. Or, you can simply use your own neural network. You can find the training information in the GitHub repository of this chapter. The file is called `trained_network.pkl`; you will also find the MNIST file (`mnist.pkl.gz`):

```
import network.network as network
import network.mnist_loader as mnist_loader
```

```
# To serialize data
import pickle
import matplotlib.pyplot as plt
import numpy as np
```

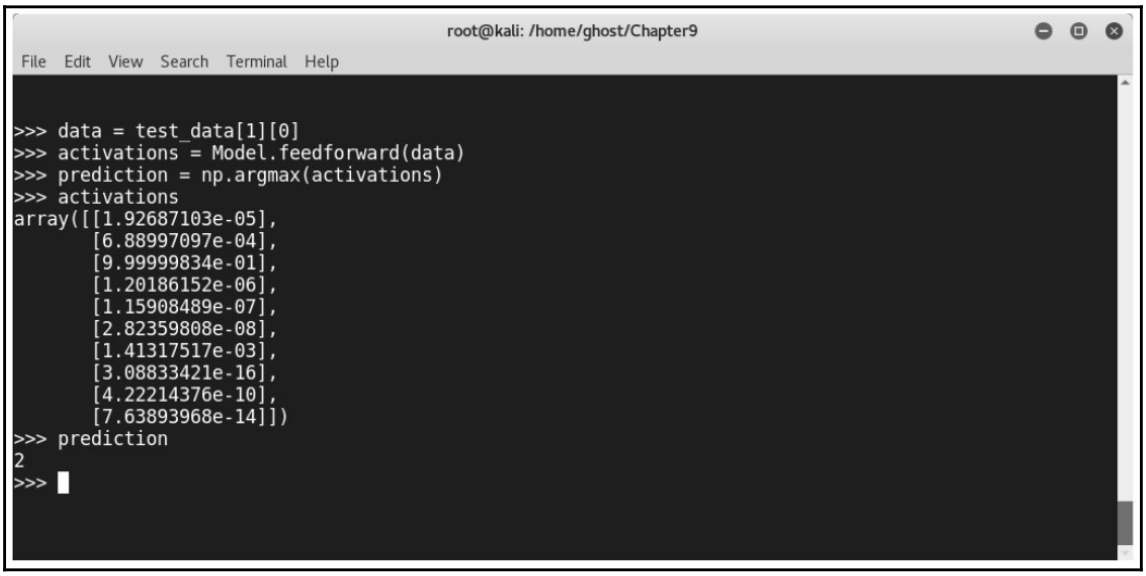
Let's check whether the model is well-trained. Load the `pickle` file. Load the data with `pickle.load()`, and identify training,validation, and testing data:

```
Model = pickle.load( open( "trained_network.pkl", "rb" ) )    trainData,
valData, testData =mnist_loader.load_data_wrapper()
```

To check digit **2**, for example, we are going to select `test_data[1][0]`:

```
>>> data = test_data[1][0]
>>> activations = Model.feedforward(data)
>>> prediction = np.argmax(activations)
```

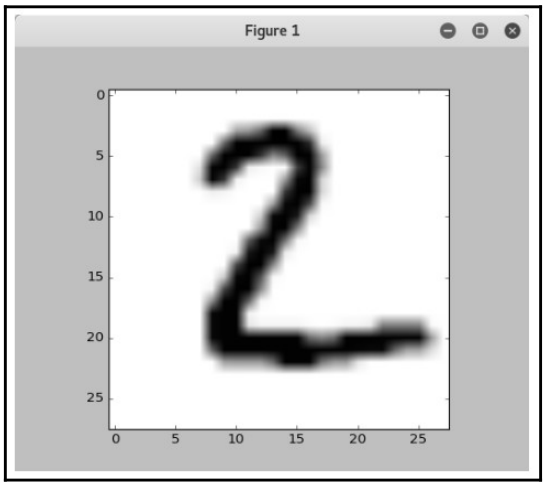
The following screenshot illustrates the preceding code:



Plot the result to check further by using `matplotlib.pyplot (plt)`:

```
>>> plt.imshow(data.reshape((28,28)), cmap='Greys')
>>> plt.show()
```

As you can see we generated the digit **2** so the model was trained well:



Everything is set up correctly. Now, we are going to attack the neural network with two types of attacks: **targeted** and **non-targeted.**

For a non-targeted attack, we are going to generate an adversarial sample and make the network give a certain output, for example, *6*:

$$Y = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In this attack, we want the neural network to think that the image entered is *6*. The target image (let's call it *X)* is a *784* dimensional vector, because the image dimension is *28×28* pixels. Our goal is to find a vector $\vec{x}$ that minimizes the cost *C*, resulting in an image that the neural network predicts as our goal label. The cost function, *C*, is defined as the following:

$$C = \frac{1}{2} \times \|Y_{goal} - \hat{y}(\vec{x})\|_2^2$$

The following code block is an implementation of a derivative function:

```
def input_derivative(net, x, y):
    """ Calculate derivatives wrt the inputs"""
    nabla_b = [np.zeros(b.shape) for b in net.biases]
    nabla_w = [np.zeros(w.shape) for w in net.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(net.biases, net.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = net.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    for l in xrange(2, net.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)
        delta = np.dot(net.weights[-l+1].transpose(), delta) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return net.weights[0].T.dot(delta)
```

To generate the adversarial sample, we need to set the goal:

```
goal = np.zeros((10, 1))
goal[n] = 1
```

Create a random image for gradient descent initialization, as follows:

```
x = np.random.normal(.5, .3, (784, 1))
```

Compute the gradient descent, as follows:

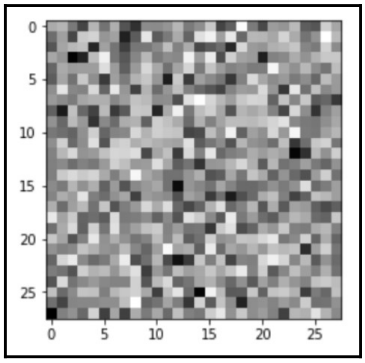
```
for i in range(steps):
        # Calculate the derivative
        d = input_derivative(net,x,goal)
        x -= eta * d
    return x
```

Now, you can generate the sample:

```
a = adversarial(net, n, 1000, 1)
x = np.round(net.feedforward(a), 2)
Print ("The input is:", str(x))
Print ("The prediction is", str(np.argmax(x)))
```

Plot the adversarial sample, as follows:

```
plt.imshow(a.reshape(28,28), cmap='Greys')
plt.show()
```



In targeted attacks, we use the same technique and the same code, but we add a new term to the cost function. So, it will be as follows:

$$C = \frac{1}{2} \times \|Y_{goal} - \hat{y}(\vec{x})\|_2^2 + \lambda \|\vec{x} - X_{target}\|_2^2$$

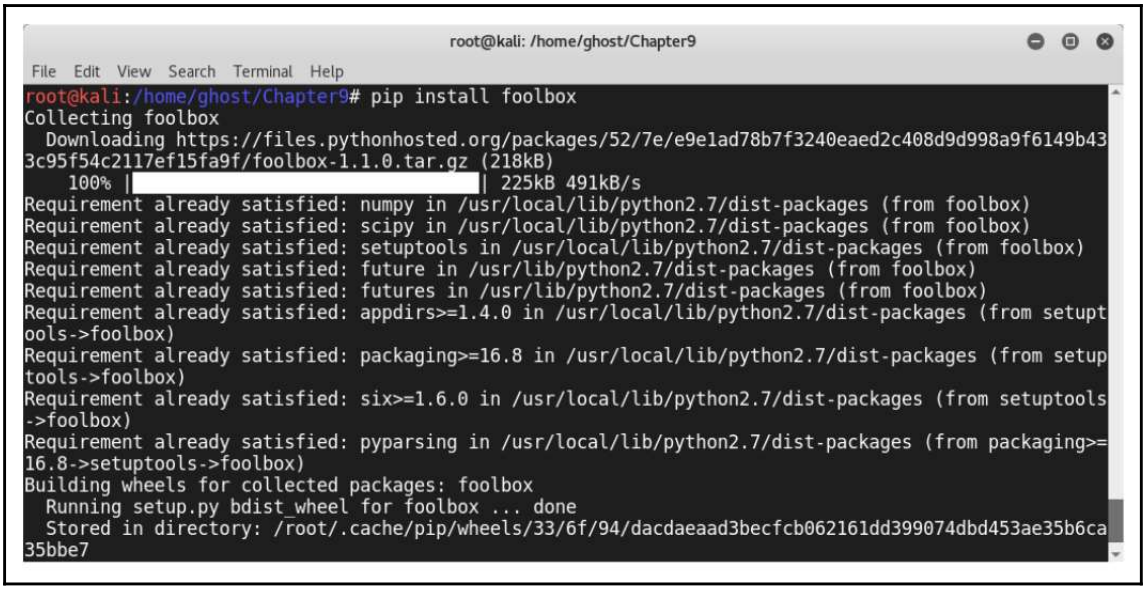
Foolbox

Foolbox is a Python toolbox to benchmark the robustness of machine learning models. It is supported by many frameworks, including the following:

- TensorFlow
- PyTorch
- Theano
- Keras
- Lasagne
- MXNet

To install Foolbox, use the `pip` utility:

```
pip install foolbox
```



The following are some Foolbox attacks:

- **Gradient-Based Attacks**: By linearizing the loss around an input, *x*
- **Gradient Sign Attack (FGSM)**: By computing the gradient, *g(x0)*, once, and then seeking the minimum step size
- **Iterative Gradient Attack**: By maximizing the loss along small steps in the gradient direction, *g(x)*
- **Iterative Gradient Sign Attack**: By maximizing the loss along small steps in the ascent direction, *sign(g(x))*
- **DeepFool L2Attack**: By computing, for each class, the minimum distance, *d(ℓ, ℓ0)*, that it takes to reach the class boundary
- **DeepFool L∞Attack**: Like L2Attack, but minimizes the *L∞-norm* instead
- **Jacobian-Based Saliency Map Attack**: By computing a saliency score for each input feature
- **Single Pixel Attack**: By setting a single pixel to white or black

To implement an attack with Foolbox, use the following:

```
import foolbox
import keras
import numpy as np
from keras.applications.resnet50 import ResNet50

keras.backend.set_learning_phase(0)
kmodel = ResNet50(weights='imagenet')
preprocessing = (np.array([104, 116, 123]), 1)
fmodel = foolbox.models.KerasModel(kmodel, bounds=(0, 255),
preprocessing=preprocessing)

image, label = foolbox.utils.imagenet_example()
attack = foolbox.attacks.FGSM(fmodel)
adversarial = attack(image[:, :, ::-1], label)
```

> If you receive the error, `ImportError('`load_weights` requires h5py.')`, solve it by installing the **h5py** library (`pip install h5py`).

To plot the result, use the following code:

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(1, 3, 1)
plt.title('Original')
plt.imshow(image / 255)
plt.axis('off')
plt.subplot(1, 3, 2)
plt.title('Adversarial')
plt.imshow(adversarial[:, :, ::-1] / 255)  # ::-1 to convert BGR to RGB
plt.axis('off')
plt.subplot(1, 3, 3)
plt.title('Difference')
difference = adversarial[:, :, ::-1] - image
plt.imshow(difference / abs(difference).max() * 0.2 + 0.5)
plt.axis('off')
plt.show()
```

Deep-pwning

Deep-pwning is a lightweight framework for experimenting with machine learning models, with the goal of evaluating their robustness against a motivated adversary. It is called the **metasploit of machine learning**. You can clone it from the GitHub repository at `https://github.com/cchio/deep-pwning`.

Don't forget to install all of the requirements:

```
pip install -r requirements.txt
```

The following are the Python libraries required to work with Deep-pwning:

- Tensorflow 0.8.0
- Matplotlib >= 1.5.1
- Numpy >= 1.11.1
- Pandas >= 0.18.1
- Six >= 1.10.0

EvadeML

EvadeML (`https://evademl.org`) is an evolutionary framework based on genetic programming, for automatically finding variants that evade detection by machine learning based malware classifiers. It was developed by the Machine Learning Group and the Security Research Group at the University of Virginia.

To download EvadeML, clone it from `https://github.com/uvasrg/EvadeML`.

To install EvadeML, you need to install these required tools:

- A modified version of pdfrw for parsing PDFs: `https://github.com/mzweilin/pdfrw`
- Cuckoo Sandbox v1.2, as the oracle: `https://github.com/cuckoosandbox/cuckoo/releases/tag/1.2`
- The target classifier PDFrate-Mimicus: `https://github.com/srndic/mimicus`
- The target classifier Hidost: `https://github.com/srndic/hidost`

To configure the project, copy the template, and configure it with an editor:

```
cp project.conf.template project.conf
Vi  project.conf
```

Before running the main program, `./gp.py`, run the centralized detection agent with predefined malware signatures, as indicated in the documentation:

```
./utils/detection_agent_server.py ./utils/36vms_sigs.pickle
```

Select several benign PDF files:

```
./utils/generate_ext_genome.py [classifier_name] [benign_sample_folder]
[file_number]
```

To add a new classifier to evade, just add a wrapper in `./classifiers/`.

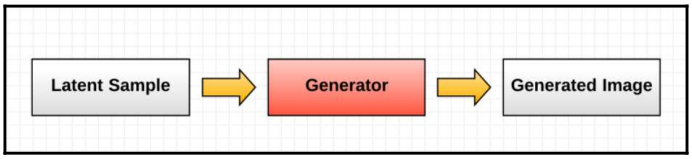
Bypassing next generation malware detectors with generative adversarial networks

In 2014, Ian Goodfellow, Yoshua Bengio, and their team, proposed a framework called the **generative adversarial network (GAN)**. Generative adversarial networks have the ability to generate images from a random noise. For example, we can train a generative network to generate images for handwritten digits from the MNIST dataset.

Generative adversarial networks are composed of two major parts: a **generator** and a **discriminator**.

The generator

The generator takes latent samples as inputs; they are randomly generated numbers, and they are trained to generate images:



For example, to generate a handwritten digit, the generator will be a fully connected network that takes latent samples and generates `784` data points, reshaping them into *28x28* pixel images (MNIST digits). It is highly recommended to use `tanh` as an activation function:

```
generator = Sequential([
Dense(128, input_shape=(100,)),
LeakyReLU(alpha=0.01),
Dense(784),
Activation('tanh')
], name='generator')
```

The discriminator

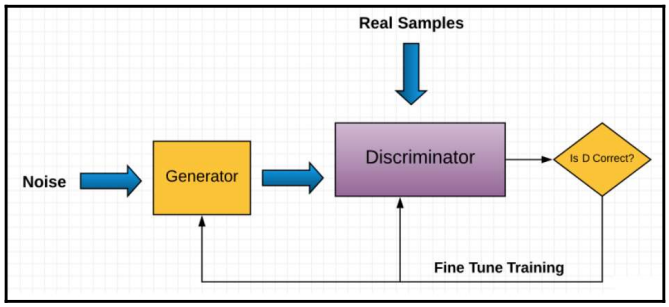
The discriminator is simply a classifier trained with supervised learning techniques to check if the image is real (`1`) or fake (`0`). It is trained by both the MNIST dataset and the generator samples. The discriminator will classify the MNIST data as real, and the generator samples as fake:

```
discriminator = Sequential([
Dense(128, input_shape=(784,)),
LeakyReLU(alpha=0.01),
Dense(1),
Activation('sigmoid')], name='discriminator')
```

By connecting the two networks, the generator and the discriminator, we produce a generative adversarial network:

```
gan = Sequential([
generator,
discriminator])
```

This is a high-level representation of a generative adversarial network:



To train the GAN, we need to train the generator (the discriminator is set as non-trainable in further steps); in the training, the back-propagation updates the generator's weights to produce realistic images. So, to train a GAN, we use the following steps as a loop:

- Train the discriminator with the real images (the discriminator is trainable here)
- Set the discriminator as non-trainable
- Train the generator

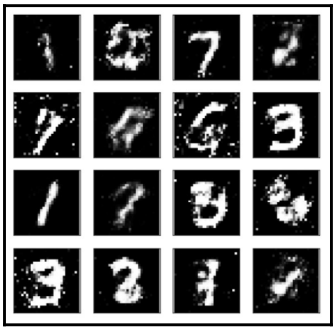
The training loop will occur until both of the networks cannot be improved any further.

To build a GAN with Python, use the following code:

```
import pickle as pkl
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
batch_size = 100
epochs = 100
samples = []
losses = []
saver = tf.train.Saver(var_list=g_vars)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for e in range(epochs):
        for ii in range(mnist.train.num_examples//batch_size):
            batch = mnist.train.next_batch(batch_size)
            batch_images = batch[0].reshape((batch_size, 784))
            batch_images = batch_images*2 - 1

            batch_z = np.random.uniform(-1, 1, size=(batch_size, z_size))
            _ = sess.run(d_train_opt, feed_dict={input_real: batch_images,
input_z: batch_z})
            _ = sess.run(g_train_opt, feed_dict={input_z: batch_z})
        train_loss_d = sess.run(d_loss, {input_z: batch_z, input_real:
batch_images})
        train_loss_g = g_loss.eval({input_z: batch_z})
        print("Epoch {}/{}...".format(e+1, epochs),
              "Discriminator Loss: {:.4f}...".format(train_loss_d),
              "Generator Loss: {:.4f}".format(train_loss_g))
        losses.append((train_loss_d, train_loss_g))
        sample_z = np.random.uniform(-1, 1, size=(16, z_size))
        gen_samples = sess.run(
                        generator(input_z, input_size,
n_units=g_hidden_size, reuse=True, alpha=alpha),
                        feed_dict={input_z: sample_z})
        samples.append(gen_samples)
```

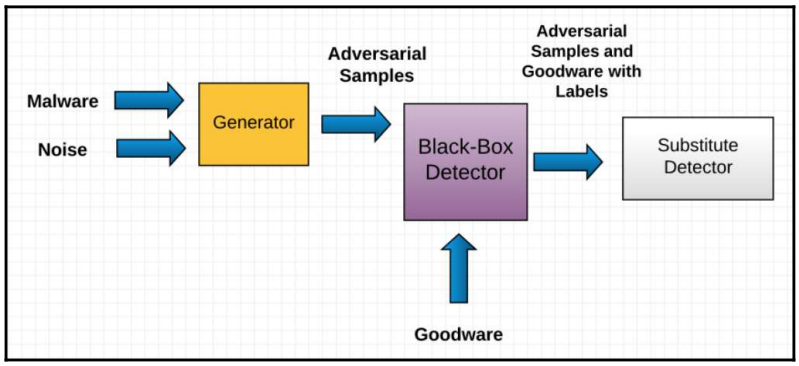
```
        saver.save(sess, './checkpoints/generator.ckpt')
with open('train_samples.pkl', 'wb') as f:
    pkl.dump(samples, f)
```



To build a GAN with Python, we are going to use NumPy and TensorFlow.

MalGAN

To generate malware samples to attack machine learning models, attackers are now using GANs to achieve their goals. Using the same techniques we discussed previously (a generator and a discriminator), cyber criminals perform attacks against next-generation anti-malware systems, even without knowing the machine learning technique used (black box attacks). One of these techniques is MalGAN, which was presented in a research project called, *Generating Adversarial Malware Examples for Black Box Attacks Based on GAN*, conducted by Weiwei Hu and Ying Tan from the Key Laboratory of Machine Perception (MOE) and the Department of Machine Intelligence. The architecture of MalGAN is as follows:



The generator creates adversarial malware samples by taking malware (feature vector *m*) and a noise vector, *z*, as input. The substitute detector is a multilayer, feed-forward neural network, which takes a program feature vector, *X*, as input. It classifies the program between a benign program and malware.

To train the generative adversarial network, the researchers used this algorithm:

```
While not converging do:
    Sample a minibatch of Malware M
    Generate adversarial samples M' from the generator
    Sample a minibatch of Goodware B
    Label M' and B using the detector
    Update the weight of the detector
    Update the generator weights
End while
```

Many of the samples generated may not be valid PE files. To preserve mutations and formats, the systems required a sandbox to ensure that functionality was preserved.

Generative adversarial network training cannot simply produce great results; that is why many hacks are needed to achieve better results. Some tricks were introduced by Soumith Chintala, Emily Denton, Martin Arjovsky, and Michael Mathieu, to obtain improved results:

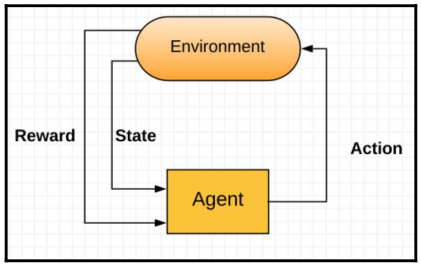
- Normalizing the images between *-1* and *1*
- Using a max log, *D*, as a loss function, to optimize *G* instead of min (*log 1-D*)
- Sampling from a Gaussian distribution, instead of a uniform distribution
- Constructing different mini-batches for real and fake
- Avoiding ReLU and MaxPool, and using LeakyReLU and Average Pooling instead
- Using **Deep Convolutional GAN** (**DCGAN**), if possible
- Using the `ADAM` optimizer

Bypassing machine learning with reinforcement learning

In the previous technique, we noticed that if we are generating adversarial samples, especially if the outcomes are binaries, we will face some issues, including generating invalid samples. Information security researchers have come up with a new technique to bypass machine learning anti-malware systems with reinforcement learning.

Reinforcement learning

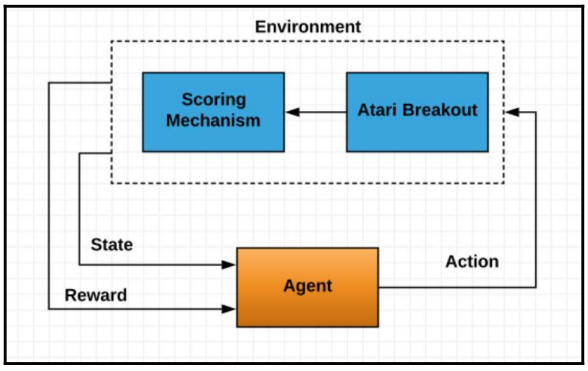
Previously (especially in the first chapter), we explored the different machine learning models: supervised, semi-supervised, unsupervised, and reinforcement models. Reinforcement machine learning models are important approaches to building intelligent machines. In reinforcement learning, an agent learns through experience, by interacting with an environment; it chooses the best decision based on a state and a reward function:



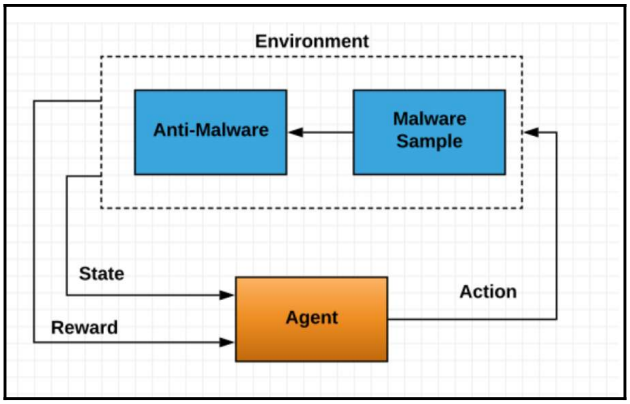
A famous example of reinforcement learning is the AI-based Atari Breakout. In this case, the environment includes the following:

- The ball and the bricks
- The moving paddle (left or right)
- The reward for eliminating the bricks

The following figure illustrates a high overview of the reinforcement model used to teach the model how to play Atari Breakout:



With the Atari Breakout environment as an analogy to learn how to avoid anti-malware systems, our environment will be as follows:



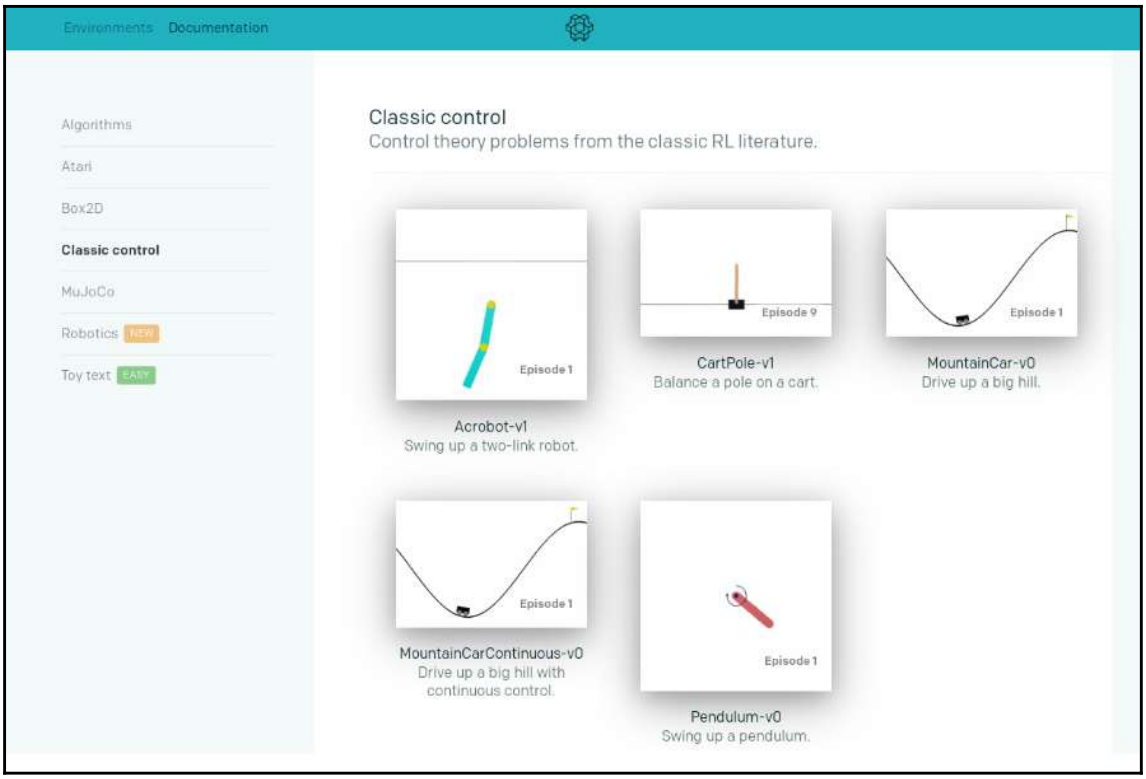
For the agent, it takes the environment state (general file information, header information, imported and exported functions, strings, and so on) to optimize its performance and the reward input from the antivirus reports, and result actions (creating entry points and new sections, modifying sections and so on). In other words, to perform and learn the agent is taking two inputs (States and rewards).

As an implementation of the concepts we've discussed, information security professionals worked on an OpenAI environment, to build a malware that can escape detection using reinforcement learning techniques. One of these environments is **Gym-malware**. This great environment was developed by endgame.

OpenAI gym contains an open source Python framework, developed by a nonprofit AI research company called OpenAI (`https://openai.com/`) to develop and evaluate reinforcement learning algorithms. To install OpenAI Gym, use the following code (you'll need to have Python 3.5+ installed):

```
git clone https://github.com/openai/gym
cd gym
pip install -e
```

OpenAI Gym is loaded pre-made environments. You can check all of the available environments at `http://gym.openai.com/envs/`:



To run an environment with Python, you can use the following code; in this snippet, I chose the `CartPole-v0` environment:

```
import gym
 env = gym.make('CartPole-v0')
 env.reset()
 for _ in range(1000): # run for 1000 steps
    env.render()
    action = env.action_space.sampe() # pick a random action
    env.step(action) # take action
```

To use the Gym-malware environment, you will need to install Python 3.6 and a library called Instrument Executable Formats, aptly named `LIEF`. You can add it by typing:

```
pip install
https://github.com/lief-project/LIEF/releases/download/0.7.0/linux_lief-0.7
.0_py3.6.tar.gz
```

Download Gym-malware from `https://github.com/endgameinc/gym-malware`. `Move the installed Gym-malware environment to gym_malware/gym_malware/envs/utils/samples/.`

To check whether you have the samples in the correct directory, type the following:

```
python test_agent_chainer.py
```

The actions available in this environment are as follows:

- `append_zero`
- `append_random_ascii`
- `append_random_bytes`
- `remove_signature`
- `upx_pack`
- `upx_unpack`
- `change_section_names_from_list`
- `change_section_names_to random`
- `modify_export`
- `remove_debug`
- `break_optional_header_checksum`

Summary

In this chapter, we continued our journey of learning how to bypass machine learning models. In the previous chapter, we discovered adversarial machine learning; in this continuation, we explored adversarial deep learning and how to fool deep learning networks. We looked at some real-world cases to learn how to escape anti-malware systems by using state of the art techniques. In the next and last chapter, we are going to gain more knowledge, learning how to build robust models.

Questions

1. What are the components of generative adversarial networks?
2. What is the difference between a generator and a discriminator?
3. How can we make sure that the malware adversarial samples are still valid when we are generating them?
4. Do a bit of research, then briefly explain how to detect adversarial samples.
5. What distinguishes reinforcement learning from deep learning?
6. What is the difference between supervised and reinforcement learning?
7. How does an agent learn in reinforcement learning?

Further reading

The following resources include a great deal of information:

- *Explaining and Harnessing Adversarial Samples*: `https://arxiv.org/pdf/1412.6572.pdf`
- *Delving Into Transferable Adversarial Examples and Black Box Attacks*: `https://arxiv.org/pdf/1611.02770.pdf`
- *Foolbox - a Python toolbox to benchmark the robustness of machine learning models*: `https://arxiv.org/pdf/1707.04131.pdf`
- *The Foolbox* GitHub: `https://github.com/bethgelab/foolbox`
- *Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN*: `https://arxiv.org/pdf/1702.05983.pdf`
- *Malware Images: Visualization and Automatic Classification*: `https://arxiv.org/pdf/1702.05983.pdf`
- *SARVAM: Search And RetrieVAl of Malware*: `http://vision.ece.ucsb.edu/sites/vision.ece.ucsb.edu/files/publications/2013_sarvam_ngmad_0.pdf`
- *SigMal: A Static Signal Processing Based Malware Triage*: `http://vision.ece.ucsb.edu/publications/view_abstract.cgi?416`