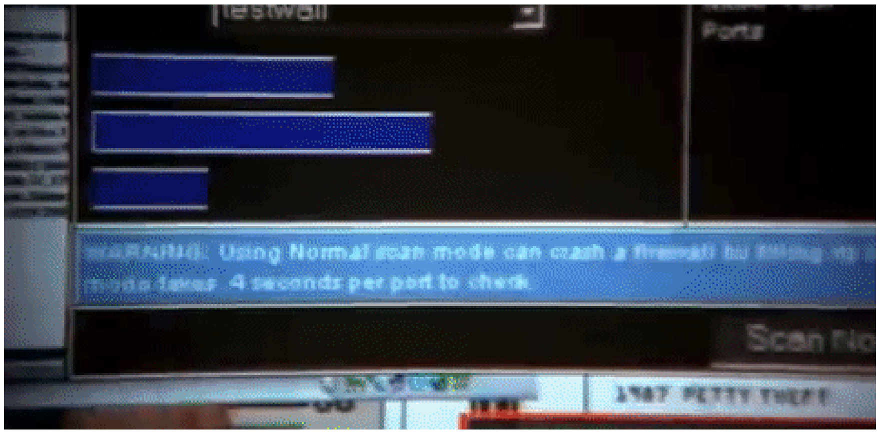
Adam Geitgey Follow
Interested in computers and machine learning. Likes to write about it.
Aug 16 · 11 min read

Machine Learning is Fun Part 8: How to Intentionally Trick Neural Networks

A Look into the Future of Hacking

This article is part of a series. Check out the full series: Part 1, Part 2, Part 3, Part 4, Part 5, Part 6, Part 7 and Part 8!

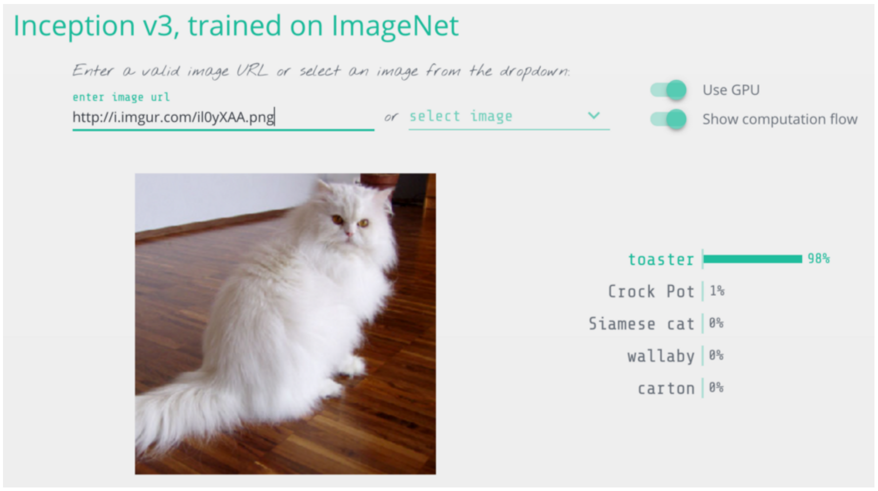
Almost as long as programmers have been writing computer programs, computer hackers have been figuring out ways to exploit those programs. Malicious hackers take advantage of the tiniest bugs in programs to break into systems, steal data and generally wreak havoc.


Ports
WARNING: Using Normal scan mode can crash a firewall
4 seconds per port to check
Scan

100% Real Hackers™

But systems powered by deep learning algorithms should be safe from human interference, right? How is a hacker going to get past a neural network trained on terabytes of data?

It turns out that even the most advanced deep neural networks can be easily fooled. With a few tricks, you can force them into predicting whatever result you want:


Inception v3, trained on ImageNet
Enter a valid image URL or select an image from the dropdown.
enter image url
http://i.imgur.com/il0yXAA.png
or
select image
Use GPU
Show computation flow
toaster 98%
Crock Pot 1%
Siamese cat 0%
wallaby 0%
carton 0%

I modified this cat picture so it would be recognized as a toaster.

So before you launch a new system powered by deep neural networks, let's learn exactly how to break them and what you can do to protect yourself from attackers.

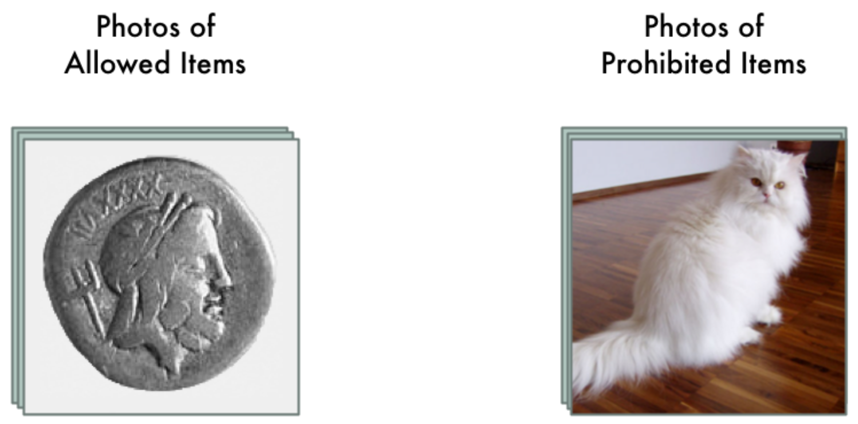
Neural Nets as Security Guards

Let's imagine that we run an auction website like Ebay. On our website, we want to prevent people from selling prohibited items—things like live animals.

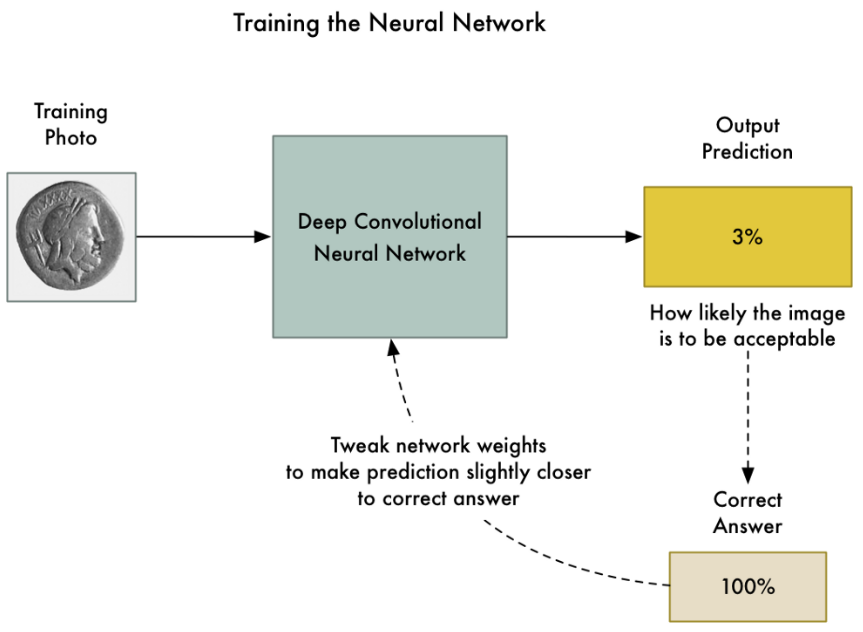
Enforcing these kinds of rules are hard if you have millions of users. We could hire hundreds of people to review every auction listing by hand, but that would be expensive. Instead, we can use deep learning to automatically check auction photos for prohibited items and flag the ones that violate the rules.

This is a typical image classification problem. To build this, we'll train a deep convolutional neural network to tell prohibited items apart from allowed items and then we'll run all the photos on our site through it.

First, we need a data set of thousands of images from past auction listings. We need images of both allowed and prohibited items so that we can train the neural network to tell them apart:

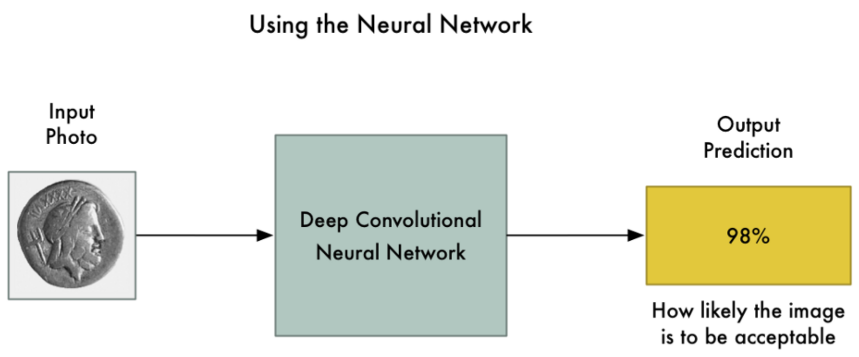


To train then neural network, we use the standard *back-propagation* algorithm. This is an algorithm were we pass in a training picture, pass in the expected result for that picture, and then walk back through each layer in the neural network adjusting their weights slightly to make them a little better at producing the correct output for that picture:


Training the Neural Network
Training Photo
Deep Convolutional Neural Network
Output Prediction
3%
How likely the image is to be acceptable
Correct Answer
100%
Tweak network weights to make prediction slightly closer to correct answer

We repeat this thousands of times with thousands of photos until the model reliably produces the correct results with an acceptable accuracy.

The end result is a neural network that can reliably classify images:

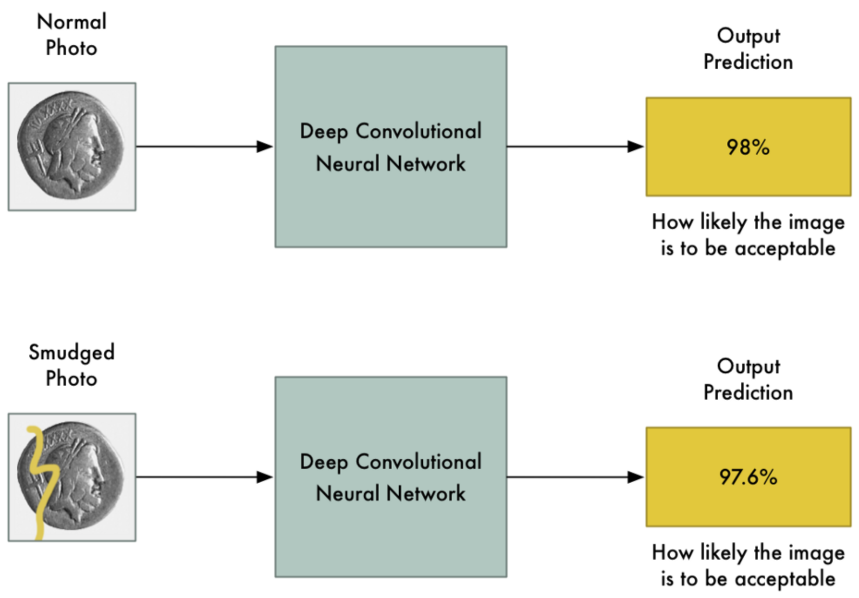

Using the Neural Network
Input Photo
Deep Convolutional Neural Network
Output Prediction
98%
How likely the image is to be acceptable

Note: If you want more detail on how convolution neural networks recognize objects in images, check out Part 3.

But things are not as reliable as they seem…

Convolutional neural networks are powerful models that consider the entire image when classifying it. They can recognize complex shapes and patterns no matter where they appear in the image. In many image recognition tasks, they can equal or even beat human performance.

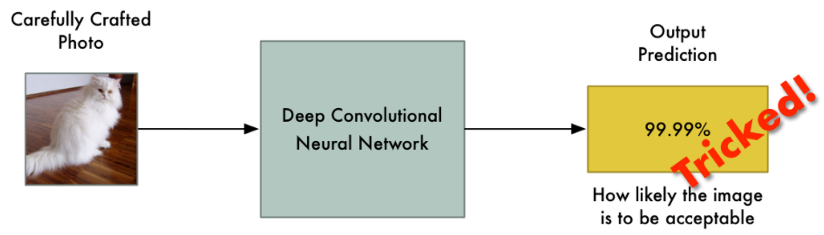
With a fancy model like that, changing a few pixels in the image to be darker or lighter shouldn't have a big effect on the final prediction, right? Sure, it might change the final likelihood slightly, but it shouldn't flip an image from "prohibited" to "allowed".


Normal Photo
Deep Convolutional Neural Network
Output Prediction
98%
How likely the image is to be acceptable
Smudged Photo
Deep Convolutional Neural Network
Output Prediction
97.6%
How likely the image is to be acceptable

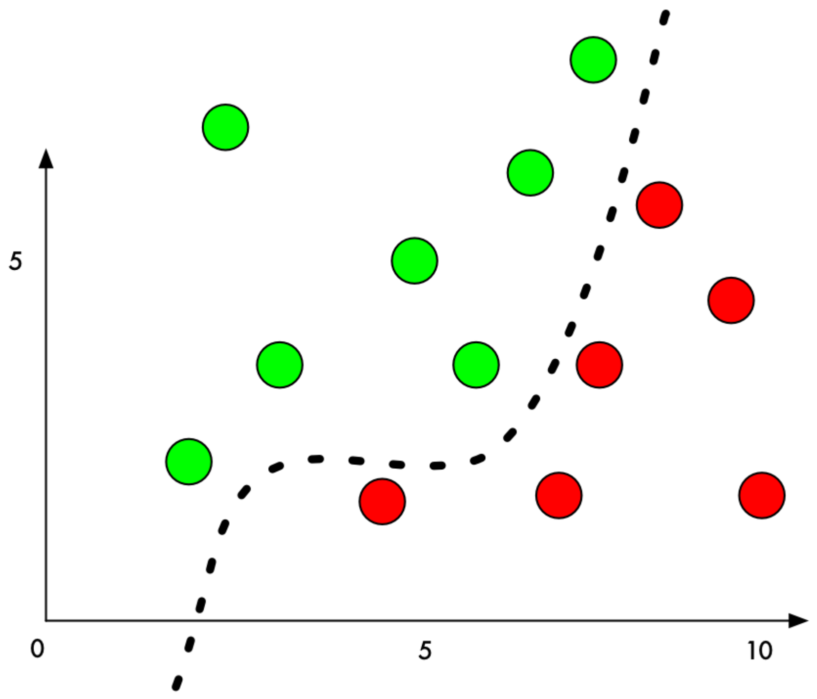
What we expect: Small changes to the input photo should only cause small changes to the final prediction.

But in a famous paper in 2013 called Intriguing properties of neural networks, it was discovered that this isn't always true. If you know *exactly which pixels to change* and *exactly how much to change them*, you can intentionally force the neural network to predict the wrong output for a given picture without changing the appearance of the picture very much.

That means we can intentionally craft a picture that is clearly a prohibited item but which completely fools our neural network:


Carefully Crafted Photo
Deep Convolutional Neural Network
Output Prediction
99.99%
Tricked!
How likely the image is to be acceptable

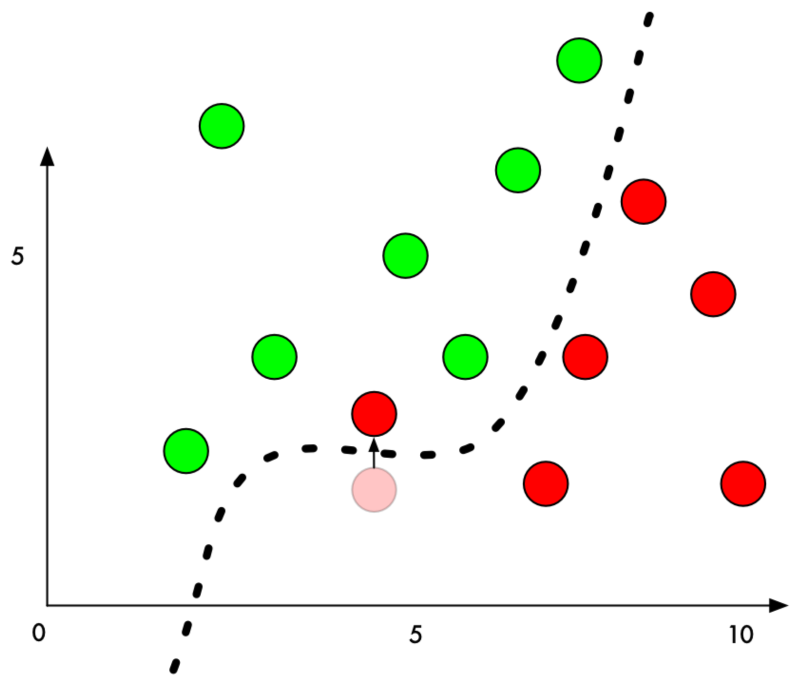
Why is this? A machine learning classifier works by finding a dividing line between the things it's trying to tell apart. Here's how that looks on a graph for a simple two-dimensional classifier that's learned to separate green points (acceptable) from red points (prohibited):



Right now, the classifier works with 100% accuracy. It's found a line that perfectly separates all the green points from the red points.

But what if we want to trick it into mis-classifying one of the red points as a green point? What's the minimum amount we could move a red point to push it into green territory?

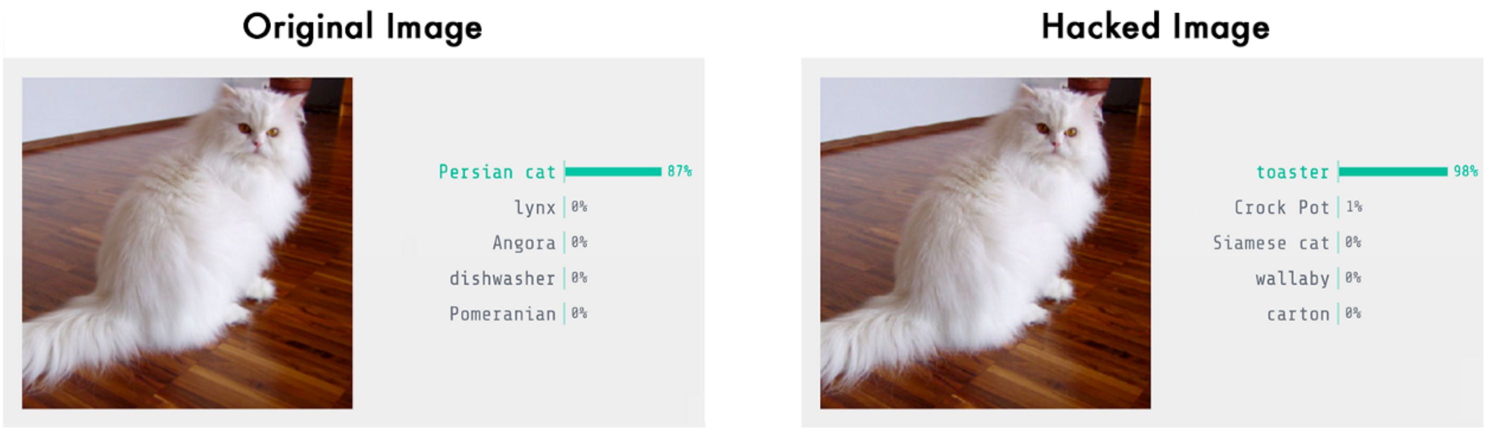
If we add a small amount to the Y value of a red point right beside the boundary, we can just barely push it over into green territory:



So to trick a classifier, we just need to know which direction to nudge the point to get it over the line. And if we don't want to be too obvious about being nefarious, ideally we'll move the point as little as possible so it just looks like an honest mistake.

In image classification with deep neural networks, each "point" we are classifying is an entire image made up of thousands of pixels. That gives us *thousands* of possible values that we can tweak to push the point over the decision line. And if we make sure that we tweak the pixels in the image in a way that isn't too obvious to a human, we can fool the classifier without making the image look manipulated.

In other words, we can take a real picture of one object and change the pixels very slightly so that the image completely tricks the neural network into thinking that the picture is something else—and we can control exactly what object it detects instead:



Turning a cat into a toaster. Image detection results from the Keras.js web-based demo

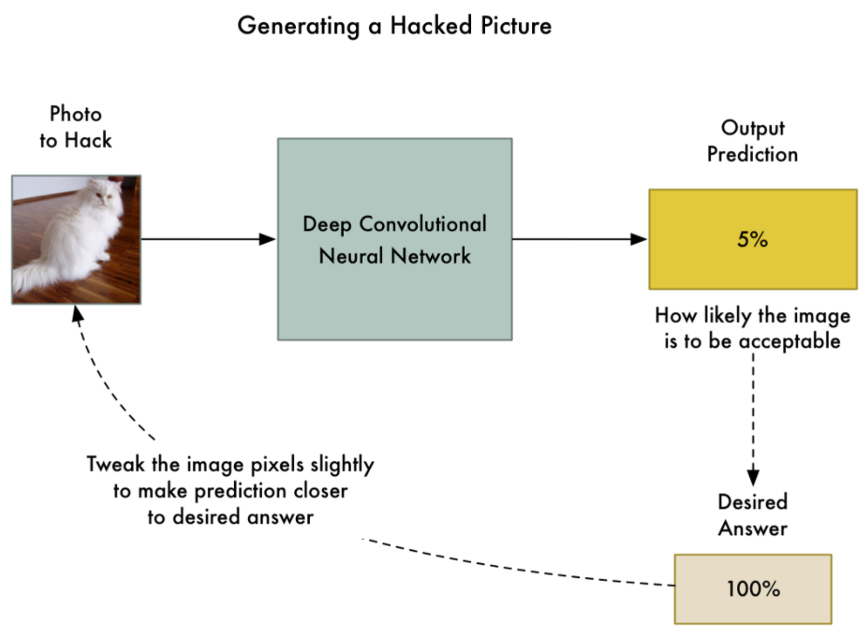
How to Trick a Neural Network

We've already talked about the basic process of training a neural network to classify photos:

1. Feed in a training photo.
2. Check the neural network's prediction and see how far off the is from the correct answer.
3. Tweak the weighs of each layer in the neural network using back-propagation to make the final prediction slightly closer to the correct answer.
4. Repeat steps 1–3 a few thousand times with a few thousand different training photos.

But what if instead of tweaking the weights of the layers of the neural network, we instead tweaked the input image itself until we get the answer we want?

So let's take the already-trained neural network and "train" it again. But let's use back-propagation to adjust the ***input image*** instead of the neural network layers:

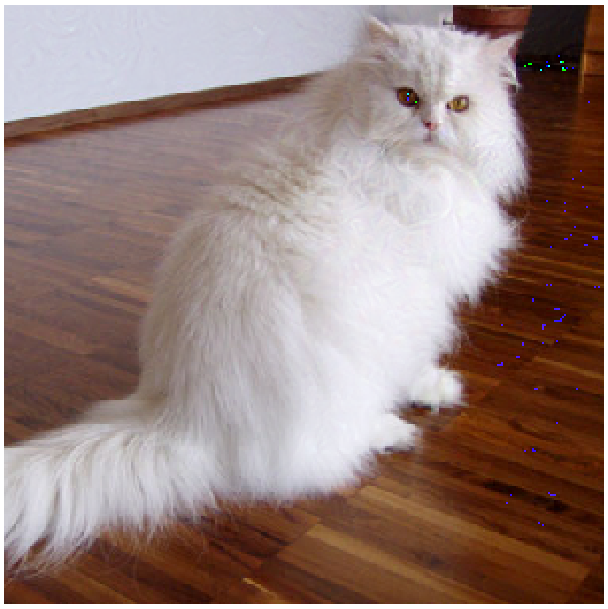

Generating a Hacked Picture
Photo to Hack
Deep Convolutional Neural Network
Output Prediction
5%
How likely the image is to be acceptable
Desired Answer
100%
Tweak the image pixels slightly to make prediction closer to desired answer

So here's the new algorithm:

1. Feed in the photo that we want to hack.
2. Check the neural network's prediction and see how far off the is from the ***answer we want to get*** for this photo.
3. Tweak our photo using back-propagation to make the final prediction slightly closer to the answer we want to get.
4. Repeat steps 1–3 a few thousand times with ***the same photo*** until the network gives us the answer we want.

At end of this, we'll have an image that fools the neural network without changing anything inside the neural network itself.

The only problem is that by allowing any single pixel to be adjusted without any limitations, the changes to the image can be drastic enough that you'll see them. They'll show up as discolored spots or wavy areas:


A hacked image with no constraints on how much a pixel can be tweaked. You can see green discolored spots around the cat and wavy patterns in the white wall.

To prevent these obvious distortions, we can add a simple constraint to our algorithm. We'll say that no single pixel in the hacked image can ever be changed by more than a tiny amount from the original image—let's say something like 0.01%. That forces our algorithm to tweak the image in a way that still fools the neural network without it looking too different from the original image.

Here's what the generated image looks like when we add that constraint:

A hacked image generated with a limit placed on how much any single pixel can be changed.

Even though that image looks the same to us, it still fools the neural network!

Let's Code It

To code this, first we need a pre-trained neural network to fool. Instead of training one from scratch, let's use one created by Google.

Keras, the popular deep learning framework, comes with several pre-trained neural networks. We'll use its copy of Google's Inception v3 deep neural network that was pre-trained to detect 1000 different kinds of objects.

Here's the basic code in Keras to recognize what's in a picture using this neural network. Just make sure you have Python 3 and Keras installed before you run it:

```python
import numpy as np
from keras.preprocessing import image
from keras.applications import inception_v3

# Load pre-trained image recognition model
model = inception_v3.InceptionV3()

# Load the image file and convert it to a numpy array
img = image.load_img("cat.png", target_size=(299, 299))
input_image = image.img_to_array(img)

# Scale the image so all pixel intensities are between [-1,
input_image /= 255.
input_image -= 0.5
input_image *= 2.

# Add a 4th dimension for batch size (as Keras expects)
```

When we run it, it properly detects our image as a Persian cat:

```
$ python3 predict.py

This is a Persian_cat with 85.7% confidence!
```

Now let’s trick it into thinking that this cat is a toaster by tweaking the image until it fools the neural network.

Keras doesn’t have a built-in way to train against the input image instead of training the neural network layers, so I had to get a little tricky and code the training step manually.

Here’s the code:

```python
import numpy as np
from keras.preprocessing import image
from keras.applications import inception_v3
from keras import backend as K
from PIL import Image

# Load pre-trained image recognition model
model = inception_v3.InceptionV3()

# Grab a reference to the first and last layer of the neura
model_input_layer = model.layers[0].input
model_output_layer = model.layers[-1].output

# Choose an ImageNet object to fake
# The list of classes is available here: https://gist.githu
# Class #859 is "toaster"
object_type_to_fake = 859

# Load the image to hack
img = image.load_img("cat.png", target_size=(299, 299))
original_image = image.img_to_array(img)

# Scale the image so all pixel intensities are between [-1,
original_image /= 255.
original_image -= 0.5
original_image *= 2.

# Add a 4th dimension for batch size (as Keras expects)
original_image = np.expand_dims(original_image, axis=0)

# Pre-calculate the maximum change we will allow to the ima
# We'll make sure our hacked image never goes past this so
# A larger number produces an image faster but risks more d
max_change_above = original_image + 0.01
max_change_below = original_image - 0.01

# Create a copy of the input image to hack on
hacked_image = np.copy(original_image)

# How much to update the hacked image in each iteration
learning_rate = 0.1
```

```

# Define the cost function.
# Our 'cost' will be the likelihood out image is the target
cost_function = model_output_layer[0, object_type_to_fake]

# We'll ask Keras to calculate the gradient based on the in
# In this case, referring to "model_input_layer" will give
gradient_function = K.gradients(cost_function, model_input_

# Create a Keras function that we can call to calculate the
grab_cost_and_gradients_from_model = K.function([model_inpu
```

If we run this, it will eventually spit out an image that will fool the neural network:

```
$ python3 generated_hacked_image.py

Model's predicted likelihood that the image is a toaster:
0.00072%

[ .... a few thousand lines of training .... ]

Model's predicted likelihood that the image is a toaster:
99.4212%
```

Note: If you don't have a GPU, this might take a few hours to run. If you do have a GPU properly configured with Keras and CUDA, it shouldn't take more than a couple of minutes to run.

Now let's test the hacked image that we just made by running it through the original model again:

```
$ python3 predict.py

This is a toaster with 98.09% confidence!
```

We did it! We tricked the neural network into thinking that a cat is a toaster!

What can we do with a Hacked Image?

Created a hacked image like this is called “generating an adversarial example”. We’re intentionally crafting a piece of data so that a machine-learning model will misclassify it. It’s a neat trick, but why does this matter in the real world?

Research has show that these hacked images have some surprising properties:

1. Hacked images can still fool neural networks even when they are printed out on paper! So you can use these hacked images to fool physical cameras or scanners, not just systems where upload an image file directly.
2. Images that fool one neural network tend to fool other neural networks with entirely different designs if they were trained on similar data.

So we can potentially do a lot with these hacked images!

But there is still a big limitation with how we create these images—our attack requires direct access to the neural network itself. Because we are actually “training” against the neural network to fool it, we need a copy of it. In the real world, no company is going to let you download their trained neural network’s code, so that means we can’t attack them… Right?

Nope! Researchers have recently shown that you can train your own substitute neural network to mirror another neural network by probing it to see how it behaves. Then you can use your substitute neural network to generate hacked images that still often fool the original network! This is called a *black-box attack*.

The applications of black-box attacks are limitless. Here are some plausible examples:

- Trick self-driving cars into seeing a stop sign as a green light—this could cause car crashes!
- Trick content filtering systems into letting offensive/illegal content through.

- Trick ATM check scanners into thinking the handwriting on a check says the check is for a greater amount then it actually is (with plausible deniability if you get caught!)

And these attack methodology isn't limited to just images. You can use the same kind of approach to fool classifiers that work on other types of data. For example, you could trick virus scanners into recognizing your virus as safe code!

How can we protect ourselves against these attacks?

So now that we know it's possible to trick neural networks *(and all other machine learning models too)*, how do we defend against this?

The short answer is that no one is *entirely* sure yet. Preventing these kinds of attacks is still an on-going area of research. The best way to keep up with the latest developments is by reading the cleverhans blog maintained by Ian Goodfellow and Nicolas Papernot, two of the most influential researchers in this area.

But there are some things we do know so far:

- If you simply create lots of hacked images and include them in your training data set going forward, that seems to make your neural network more resistant to these attacks. This is called Adversarial Training and is probably the most reasonable defense to consider adopting right now.
- There is another somewhat effective approach called Defensive Distillation where you train a second model to mimic your original model. But this approach is new and rather complicated, so I wouldn't invest in this yet unless you have specialized needs.
- Pretty much every other idea researchers have tried so far has failed to be helpful in preventing these attacks.

Since we don't have any final answers yet, its worth thinking about the scenarios where you are using neural networks so that you can at least lessen the risk that this kind of attack would cause damage your business.

For example, if you have a single machine learning model as the only line of defense to grant access to a restricted resource and assume it can't be fooled, that's probably a bad idea. But if you use machine learning as a step in a process where there is still human verification, that's probably fine.

In other words, treat machine learning models in your architecture like any other component that can potentially be bypassed. Think through the implications of what would happen if a user intentionally sets out to fool them and think of ways to mitigate those scenarios.

Learning More

Want to learn more about Adversarial Examples and protecting against them?

- This area of research is only a few years old, so it's easy to get caught up by reading a few key papers: Intriguing properties of neural networks, Explaining and Harnessing Adversarial Examples, Practical Black-Box Attacks against Machine Learning and Adversarial examples in the physical world.
- Follow Ian Goodfellow and Nicolas Papernot's cleverhans blog to keep up with the latest research.
- Check out the on-going Kaggle competition that's challenging researchers to come up with new ways to defend against these attacks.

· · ·

If you liked this article, please consider **signing up for my Machine Learning is Fun! email list**. I'll only email you when I have something new and awesome to share. It's the best way to find out when I write more articles like this.

You can also follow me on Twitter at @ageitgey, email me directly or find me on linkedin. I'd love to hear from you if I can help you or your team with machine learning.