Seven Thousandths from a Gibbon: How to Fool an AI with Geometry

Antwi

Here is a panda.



It's a young giant panda in a tree. It has dark patches around its eyes, black ears and a bit of branch in the lower left corner. If you were asked what it was, you'd say "a panda" and wonder why I was asking.

In 2014, some Google researchers (Goodfellow, Shlens and Szegedy, 2015) showed this image to GoogLeNet, one of the best image recognition programs of the time. Unsurprisingly, it also said panda, though only with 57.7% confidence.

Now imagine a second picture that, to a human eye, is identical to the first. GoogLeNet identifies it as a gibbon, a kind of small ape with long arms, with 99.3% certainty.

The second picture is simply the first picture with a small calculated noise vector added to it:

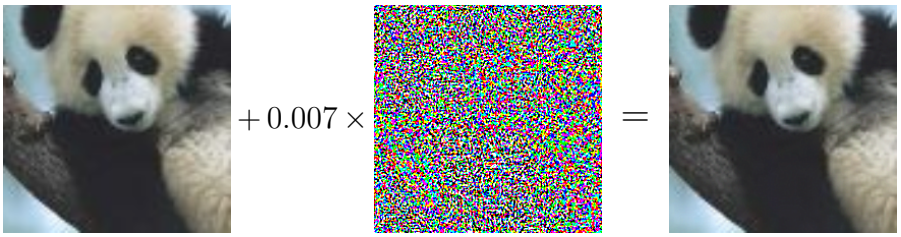


Figure 1: Images taken from Goodfellow, Shlens and Szegedy (2015).

The middle image looks like random static, but it has been calculated extremely precisely. Every value in the photo has been nudged up or down by 0.007, about the smallest change an ordinary image file can store.

Pictures engineered like this are called *adversarial examples*, a name given by the researchers who first described them (Szegedy et al., 2014). According to Goodfellow, Shlens and Szegedy (2015), the reason they work has very little to do with software glitches and almost everything to do with how geometry behaves when space has many thousands of dimensions.

A Picture is a Point

The panda photo is 224 pixels wide by 224 pixels high. Each pixel is represented by three colour numbers: red, green and blue. The entire photo is a list of:

$$224 \times 224 \times 3 = 150{,}528 \text{ numbers}$$

We can represent this list as a vector (a list of numbers treated as a single object, like coordinates of a point) $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $n = 150{,}528$. In other words, the photo is a single point in a 150,528-dimensional space, and every other possible photo of this size is another point in the same space.

Since it would be hard to imagine 150,528 dimensions, let's use a simplified model: an image with just two pixels. Each two-pixel picture is a point (x_1, x_2) in a two-dimensional plane.

Let's say we're trying to classify these two-pixel images as pandas and gibbons. The simplest classifier draws a straight line on the plane:

$$w_1 x_1 + w_2 x_2 = c$$

Given a weight vector $\mathbf{w} = (w_1, w_2)$ and threshold c, we calculate the score $s = \mathbf{w} \cdot \mathbf{x}$, the *dot product*: multiply matching entries and add them up, so $s = w_1 x_1 + w_2 x_2$. Points on one side of the line are pandas and points on the other side are gibbons.

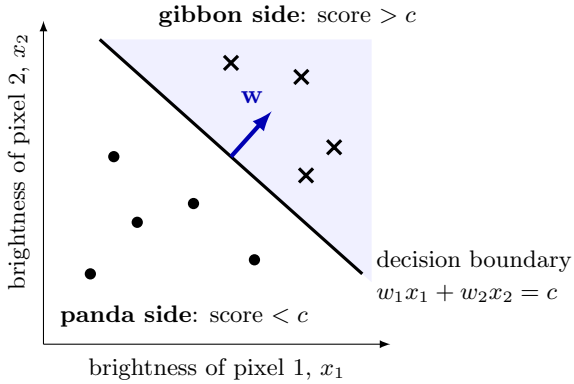


Figure 2: Each two-pixel image is a point: dots are pandas, crosses are gibbons. The arrow $\mathbf{w}$ is at right angles to the line and points the way the score increases fastest.

In higher dimensions the straight line becomes a *hyperplane*, but the mathematical rule remains identical:

$$\mathbf{w} \cdot \mathbf{x} = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \sum_{i=1}^{n} w_i x_i$$

A program that sorts inputs this way is called a *linear classifier*. During training, the computer adjusts the weight vector $\mathbf{w}$ using millions of labelled images, until the decision boundary separates the classes correctly.

The Optimisation Problem

Suppose you are an attacker who can change each coordinate x_i of an image $\mathbf{x}$ by at most ϵ. What is the largest amount you can change the classification score $\mathbf{w} \cdot \mathbf{x}$ by?

Consider a perturbation vector $\boldsymbol{\eta}$ (the Greek letter eta: the list of changes we make to each pixel) and a new image $\mathbf{x} + \boldsymbol{\eta}$, with the constraint $|\eta_i| \leq \epsilon$ for every pixel i. Using the linearity of the dot product:

$$\mathbf{w} \cdot (\mathbf{x} + \boldsymbol{\eta}) = \mathbf{w} \cdot \mathbf{x} + \mathbf{w} \cdot \boldsymbol{\eta}$$

So the only change in the score is $\mathbf{w} \cdot \boldsymbol{\eta}$.

It helps to try a small example by hand first. Take an image with four pixels and a classifier with these weights:

$$\mathbf{x} = (0.2,\ 0.9,\ 0.5,\ 0.7), \qquad \mathbf{w} = (0.3,\ -0.5,\ 0.1,\ -0.1).$$

The score is

$$\mathbf{w} \cdot \mathbf{x} = 0.06 - 0.45 + 0.05 - 0.07 = -0.41.$$

Say anything below 0 counts as a panda, so this is a panda. Now we may move each pixel by at most $\epsilon = 0.01$, and we want to push the score up as far as possible. The first weight is positive, so making x_1 bigger raises the score: we add 0.01. The second weight is negative, so making x_2 *smaller* raises the score: we subtract 0.01. Doing the same for all four pixels gives

$$\boldsymbol{\eta} = (+0.01,\ -0.01,\ +0.01,\ -0.01),$$

and the score goes up by

$$\mathbf{w} \cdot \boldsymbol{\eta} = 0.003 + 0.005 + 0.001 + 0.001 = 0.01.$$

The new score is -0.40. Still a panda, and barely any different, but notice the trick: every pixel pushed the score the same way.

In general, to maximise this change, we consider each dimension independently:

$$w_i \eta_i \leq |w_i|\,|\eta_i| \leq \epsilon |w_i|$$

Adding up the shift over all n dimensions, the maximum possible shift is:

$$\mathbf{w} \cdot \boldsymbol{\eta}_{\max} = \epsilon \sum_{i=1}^{n} |w_i| = \epsilon \|\mathbf{w}\|_1$$

So the best push is to move every pixel by the full ϵ, in the direction its weight points:

$$\boldsymbol{\eta} = \epsilon\, \mathrm{sign}(\mathbf{w}),$$

where $\mathrm{sign}(\mathbf{w})$ means replacing each weight with $+1$ if it is positive and -1 if it is negative. The sum of the sizes of the weights, ignoring signs, is called the *1-norm*, written $\|\mathbf{w}\|_1$. This is exactly what we did in the example, where $\|\mathbf{w}\|_1 = 1$ and the score rose by 0.01.

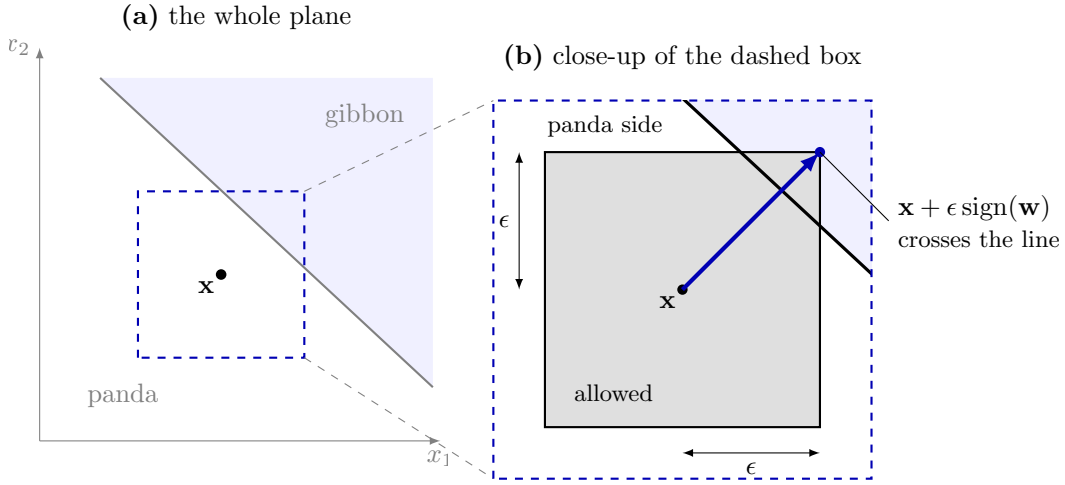


Figure 3: (a) The image $\mathbf{x}$ is on the panda side, close to the line. (b) The new image must stay inside the grey square. None of its sides reach the line, but one corner does, and the push $\epsilon\,\mathrm{sign}(\mathbf{w})$ jumps straight to it.

How Far is the Boundary?

The change in score grows with n, since there is one term for every pixel. But scores alone can mislead: doubling every weight and c leaves the line where it is, yet doubles every score. What actually matters is the distance from the image to the boundary.

The shortest distance from an image $\mathbf{x}$ to the boundary $\mathbf{w} \cdot \mathbf{x} = c$ is:

$$d = \frac{|\mathbf{w} \cdot \mathbf{x} - c|}{\|\mathbf{w}\|_2} = \frac{|\mathbf{w} \cdot \mathbf{x} - c|}{\sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}}$$

The top is how far the score is from the threshold. The bottom, written $\|\mathbf{w}\|_2$, is the length of $\mathbf{w}$, found by Pythagoras. Dividing by it undoes the scaling problem: double the weights and c, and the top and bottom both double, so d stays the same.

Using our optimal perturbation $\boldsymbol{\eta} = \epsilon\,\mathrm{sign}(\mathbf{w})$, the distance we move towards the boundary is:

$$\Delta d = \epsilon \times \frac{\sum_{i=1}^{n} |w_i|}{\sqrt{\sum_{i=1}^{n} w_i^2}} = \epsilon R$$

In the four-pixel example, the top is 1 and the bottom is $\sqrt{0.09 + 0.25 + 0.01 + 0.01} = \sqrt{0.36} = 0.6$, so $R \approx 1.67$. The push moved the picture about 0.017 towards the line, but it started $0.41/0.6 \approx 0.68$ away. No wonder it was still a panda.

The amplification ratio $R = \dfrac{\|\mathbf{w}\|_1}{\|\mathbf{w}\|_2}$ decides how much a very small pixel change is magnified in high-dimensional space. R is always between 1 and $\sqrt{n}$. It is biggest when all the weights are the same size: if every $|w_i| = m$, the top is nm and the bottom is $\sqrt{nm^2} = m\sqrt{n}$, so $R = \sqrt{n}$. For a standard $224 \times 224 \times 3$ image ($n = 150{,}528$), the maximum magnification factor is:

$$R_{\max} = \sqrt{150{,}528} \approx 388$$

There is also a geometric reason why $\sqrt{n}$ appears. The constraint $|\eta_i| \leq \epsilon$ means the new image has to stay inside a box centred on $\mathbf{x}$: a square in two dimensions (the grey square in Figure 3), a cube in three, and an n-dimensional hypercube in general. The faces of the box are always ϵ from the centre, but its corners are

$$d_{\text{corner}} = \sqrt{\epsilon^2 + \epsilon^2 + \cdots + \epsilon^2} = \epsilon\sqrt{n} \approx 0.007 \times 388 \approx 2.72$$

away. The perturbation $\boldsymbol{\eta} = \epsilon\,\text{sign}(\mathbf{w})$ always points towards one of these corners. So although a change of 0.007 in each pixel is not perceptible to humans, to the neural network, which takes all 150,528 dimensions into account at once, the image point has travelled a large distance through the space.

Beyond Linear Approximations

Many of the key building blocks of deep neural networks are piecewise linear functions, such as the ReLU, $\max(0, x)$. Designers use them because they make networks much easier to train (Goodfellow, Shlens and Szegedy, 2015).

Close to the input image $\mathbf{x}$, a curved decision boundary can be treated as if it were flat. The idea is the same as a tangent line: if you zoom in far enough on a smooth curve, it looks like a straight line, so for a small step, new value $\approx$ old value + slope $\times$ step. With many variables it works the same way, except there is one slope for each pixel: how fast the score changes if you nudge that pixel alone and leave the rest. The list of all these slopes is a vector called the *gradient*, written $\nabla f(\mathbf{x})$. So for a small change $\boldsymbol{\eta}$ to the whole image:

$$f(\mathbf{x} + \boldsymbol{\eta}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x}) \cdot \boldsymbol{\eta}$$

So the best adversarial perturbation is:

$$\boldsymbol{\eta} = \epsilon\,\text{sign}(\nabla f(\mathbf{x}))$$

Compare this with the linear classifier: the gradient $\nabla f(\mathbf{x})$ is playing exactly the role the weights $\mathbf{w}$ played before. This is known as the *Fast Gradient Sign Method* (FGSM), introduced in the same paper. In some sense, the same properties that make deep networks easy to train are what make them vulnerable to perturbations in high dimensions.

Conclusion

So the panda becomes a gibbon not because of a software glitch, but because of geometry. A change of 0.007 in each pixel is far too small for us to see, but a classifier adds up all 150,528 of them, and the image travels a long way through the space the classifier works in. This isn't just a trick on a screen either: a 3D-printed turtle with a specially designed pattern on its shell can fool a computer vision system into seeing a rifle (Athalye et al., 2018).

As AI models are used in important areas such as self-driving cars, medical diagnosis and security, it becomes increasingly important to understand the geometry of high-dimensional space.

Acknowledgements. Thanks to ChatGPT for drawing the diagrams, since my LaTeX skills aren't quite there yet. I asked it whether they were all correct, and it said it was 99.3% sure.