DS 102 Data, Inference, and Decisions **Spring 2020**

Lecture 27: Robustness and Distribution Shift

Lecturer: Jacob Steinhardt

Today we will discuss robustness of machine learning models. There are generally two types of robustness that people study:

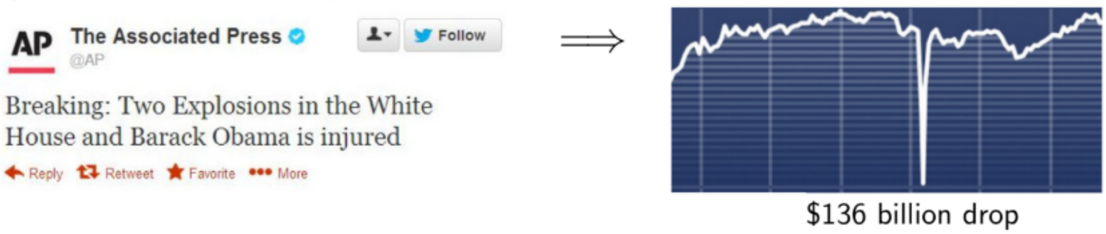
1. Robustness against adversaries that are actively attempting to attack or fool the model
2. Robustness against things changing in the world

Today, we will focus on the first type of robustness, which is particularly important for applications like face recognition and spam detection. The solutions for robustness to changes in the world which are not adversarial—often known as **distribution shift**—are often similar to those for adversarial robustness, although it is also its own active area of research.

27.1 Examples of Adversarial Attacks

Several recent examples highlight the importance of adversarial robustness.

In 2013, Syrian hackers used the Associated Press's twitter feed falsely report "Two Explosions in the White House and Barack Obama is injured," leading to a $136 billion drop in the U.S. stock market. More recently, bots influenced the U.S. and other elections by creating fake twitter accounts to mutually retweet each other. These fake retweets affected trending topics, fooling Twitter's algorithms (and legitimate Twitter users) into thinking particular topics were more important.

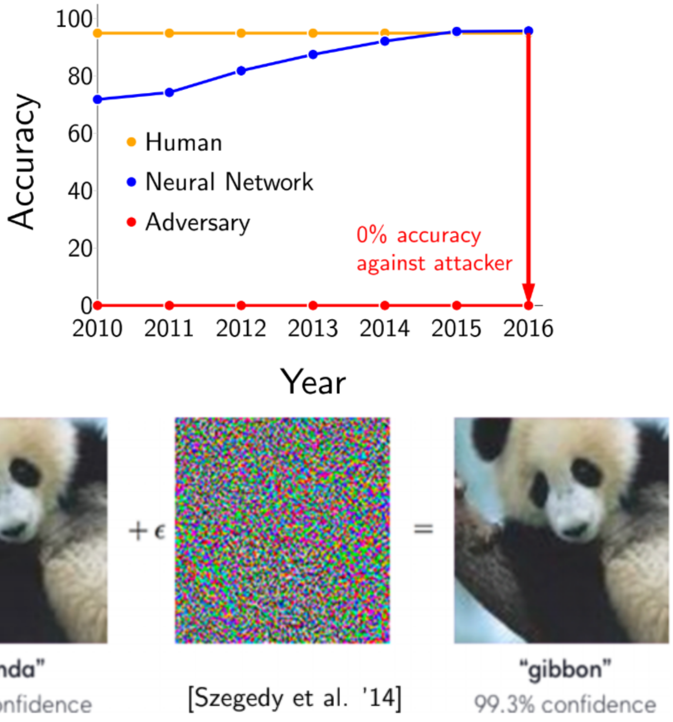


AP The Associated Press
@AP

Breaking: Two Explosions in the White House and Barack Obama is injured

Figure 27.1: Hackers' fake tweet via the Associated Press temporarily caused a massive drop in the U.S. stock market.

Adversarial robustness is a particularly relevant when we think about neural network models. Deep learning is state-of-the-art in many domains, for instance computer vision. However, these neural network systems, despite their success at achieving high accuracy, are incredibly fragile. For instance, slightly modifying an image in an adversarial way can cause state-of-the-art deep learning models for vision tasks to achieve essentially 0% accuracy, even when the modifications are not perceived by humans (see Figure 27.2). Adversarial examples are persistent despite hundreds of papers trying to avoid them.



[Szegedy et al. '14]

Figure 27.2: State-of-the-art neural networks for vision applications are extremely sensitive to adversarial attacks.

27.2 Why do Adversarial Attacks Work?

Why aren't our models robust against these kinds of adversarial attacks? Most machine learning systems assume that the distributions of the training data and the test data are the same (or at least very closely related). However, there are many situations where the test data received during deployment may be very different from the training data that was collected. Attackers can easily violate this assumption that train $\approx$ test, creating vulnerabilities.

One reason that it can be so difficult to make our models robust to attacks is that simple empirical evaluation is not sufficient to evaluate robustness. Indeed, the history of the field of adversarial robustness is a testament to this fact. The phenomenon of adversarial images (like that in Figure 27.2) was discovered in 2014. In the following year, there were a few papers offering solutions, including *adversarial training* and *defensive distillation*. Within a year, several papers discovered more attacks breaking these proposed defensive solutions. There have since been hundreds of more papers demonstrating new defenses and subsequent attacks that break these defenses—it typically takes a few months for new papers to show how to break of bypass new defenses. This history shows that, when it comes to adversarial robustness, one cannot simply just try stuff and see what works—this leads to a "security arms race" that defenders often lose. Rather, new methodology is needed to evaluate robustness.

27.3 Generating Adversarial Examples

Suppose we have a neural network that we are using to perform image classification. We can think of the network as some function of its input x (a vector representing an image) and its parameters θ (the weights of the network): $f(x, \theta)$. For classification, the network gives the probability $p(y|x, \theta)$ that the image x belongs to class y. Let us denote by $\hat{y}$ the model's prediction, for instance a classification as a "panda." During training, we have some true label y^* for a given image x, and our goal is to update the parameters θ so that $\hat{y} = y^*$. Recall from Lecture 25: we update θ using backpropogation, which means we compute $\nabla_\theta \log p(y^*|x, \theta)$ and use it to try to make $\log p(y^*|x, \theta)$ large.

When we attack a trained network, we do not get to update the model parameters. Instead, the attacker's goal is to update the input x to make $\hat{y} \neq y^*$. To achieve this, the attacker computes $\nabla_x p(y^*|x, \theta)$ and uses it to try make $\log p(y^*|x, \theta)$ small. This is the high-level idea behind generating adversarial examples: we update the input instead of the weights, and do so to try to make predictions wrong instead of making them right. However, we also often want to constrain x to still be a realistic-looking image. One way to do this is to start with some real image x_0 (for example, the panda image in Figure 27.2) and constrain our new x to be close to x_0 in terms of their individual pixels:

$$|x_j - (x_0)_j| \leq \epsilon \; \forall j.$$

Note that we can write these constraints more compactly as either

$$\max_j |x_j - (x_0)_j| \leq \epsilon$$

or

$$\|x - x_0\|_\infty \leq \epsilon,$$

where the latter is called the ℓ_∞ norm.

How do we make $\log p(y^*|x, \theta)$ as wrong as possible subject to the constraints $|x_j - (x_0)_j| \leq \epsilon$? One simple way to do this is called the Fast Gradient Sign Method (FGSM):

1. Compute the gradient $g = \nabla_x \log p(y^*|x, \theta)$
2. If $g_j < 0$ set $x_j = (x_0)_j + \epsilon$
3. If $g_j > 0$ set $x_j = (x_0)_j - \epsilon$

FGSM can be written compactly as $x = x_0 - \epsilon\text{sign}(g)$, and is nice because it can be implemented in two lines of code and runs very quickly.

An improvement on FGSM is to iterate the process:

```
1. Start at $x_0$ and pick some step size $\eta < \epsilon$
2. For $t = 1, 2, \ldots$:
    • Compute $g_{t-1} = \nabla_x \log p(y^*|x_{t-1}, \theta)$
    • Update $x_t = x_{t-1} - \eta \text{sign}(g_{t-1})$
```

However, as written this algorithm has the issue that if we run for many iterations (more than $\frac{\epsilon}{\eta}$ steps), our x_t will eventually violate our constraints. We need to add a projection step that forces x_t to stay within our constraint set. This projection clips each coordinate j to be between $(x_0)_j - \epsilon$ and $(x_0)_j + \epsilon$ (and typically also clips to be between 0 and 1, since this is the range of valid pixel values). This gives us Projected Gradient Descent:

```
1. Start at $x_0$ and pick some step size $\eta < \epsilon$
2. For $t = 1, 2, \ldots$:
    • Compute $g_{t-1} = \nabla_x \log p(y^*|x_{t-1}, \theta)$
    • Update $x_t = \text{clip}(x_{t-1} - \eta \text{sign}(g_{t-1}))$
```

Another variation of this algorithm is called the Iterative Sign Method. Iterating the gradient steps tends to give slightly stronger attacks relative to FGSM, where here “stronger” means that the image can be modified less while still causing predictions to be wrong.

27.4 Defending Against Adversarial Attacks

We will begin with some loose intuition for why FGSM works for generating adversarial examples. Recall that FGSM takes $x_1 = x_0 - \epsilon \text{sign}(g_0)$, where $g_0 = \nabla_x \log p(y^*|x_0, \theta)$. We can understand $\log p(y^*|x_1, \theta)$ by looking at its Taylor expansion:

$$\begin{aligned}\log p(y^*|x_1, \theta) &\approx \log p(y^*|x_0, \theta) + \langle \nabla_x \log p(y^*|x_0, \theta), x_1 - x_0 \rangle \\ &= \log p(y^*|x_0, \theta) + \langle g, -\epsilon \text{sign}(g) \rangle \\ &= \log p(y^*|x_0, \theta) - \epsilon \sum_j |g_j|.\end{aligned}$$

Roughly how big is $\sum_j |g_j|$? When we are working in d dimensions, the sum $\sum_{j=1}^d |g_j|$ has d terms. If we assume that all of the g_j's have roughly the same magnitude (call it c), then can roughly say

$$\log p(y^*|x_1, \theta) \approx \log p(y^*|x_0, \theta) - \epsilon c d,$$

which suggests we can change the model's predictions quite a lot even if ϵ is very small. It turns out that this scaling with the dimension d is not quite the right intuition, since in high dimensions each coordinate of the gradient also gets smaller. A slightly better calculation is that $|g_j| \approx c/\sqrt{d}$, so $\sum_j |g_j| \approx c\sqrt{d}$. This gives $\epsilon c\sqrt{d}$ as the rough amount that FGSM can alter things. Again, this is likely large enough to have an impact even when ϵ is small.

Given than FGSM and Projected Gradient Descent seem to work well, how can we defend against attacks like these? **Adversarial training** is one key idea for making models robust against attacks. Roughly speaking, adversarial training works by running the attack at train time and adding the attacked images to the training set. For example, a common instantiation of adversarial training is:

- For each (x, y^*) in the training data:
  1. Generate $x' = \text{attack}(x, y^*)$
  2. Update θ based on (x', y^*)
  3. Possible also update θ based on (x, y^*)

Using adversarial training can lead to models that are more robust. However, if too few gradient steps are used in the optimizer that generates the adversarial examples, the model tends to just learn to fool the optimizer instead of learning to be truly robust (see Figure 27.3). Since, in adversarial training, we are explicitly training against a particular attack, if there is any easy way for the model to fool the attack without actually learning to be robust, it will tend to do that. This phenomenon is known as **gradient masking**.

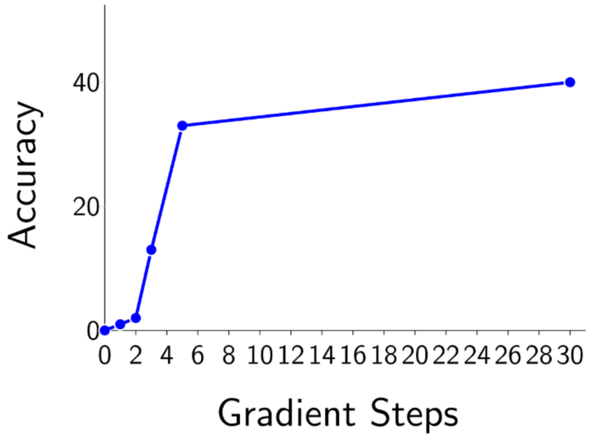


Figure 27.3: Plot relating the number of gradient steps used to generate adversarial examples to model accuracy for an example neural network on a vision task. This plot demonstrates the gradient masking phenomenon, wherein robustness (high accuracy) is only achieved when enough gradient steps are used.

Another way to gain intuition for this gradient masking phenomenon is to visualize the images that maximally excite different neurons in a trained neural network model (for example, using the library Lucid). As shown in Figure 27.4, the visualizations for the robust (adversarially-trained) model are clearer than for the "regular" model. Indeed, the clarity of the visualized images tends to increase as we increase the number of gradient steps in the optimizer used to generate the attacks during adversarial training.

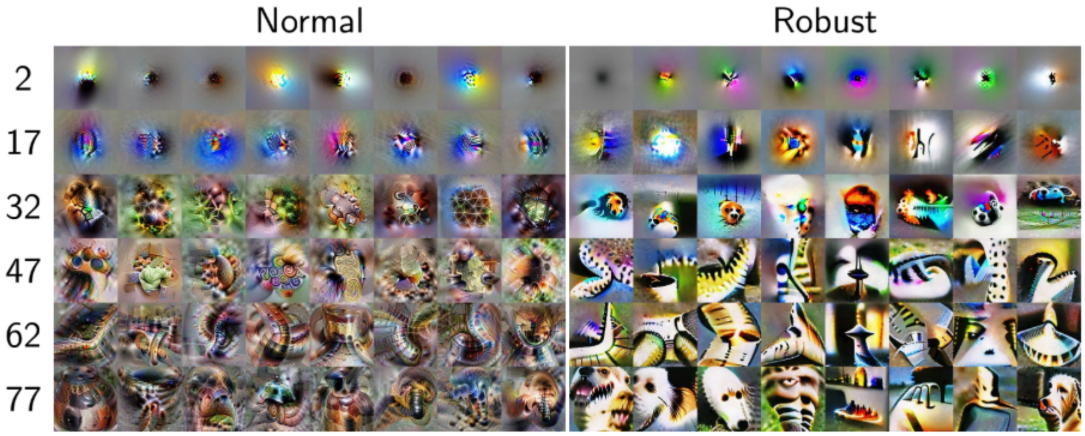


Figure 27.4: Images that maximally excite different neurons in normal and adversarially-trained neural network models.

There is an important caveat to keep in mind here: so far we have discussed algorithms that only protect against one specific type attack. Ideally, we would like our models to be robust to a lot of different attacks. One approach is to design many different attacks ourselves, do adversarial training against each of those attacks, and then evaluate how robust our model is to all of these attacks. However, this is in some sense cheating, because we are implicitly assuming that we already know the full suite of attacks against which our model needs to be robust. Something slightly more realistic is to train against some small number of different attacks, and then test against types of attacks that were not included at train time. This probes how robust the model is to unforeseen attacks. There is still currently a big gap between how well models can do on foreseen and unforeseen attacks (e.g. some approaches can do roughly half as well on unforeseen attacks). This is an active area of research, including here at UC Berkeley.