CIS 7000-005: Debugging Data & Models Fall 2022

Adversarial — September 15

Prof. Eric Wong

Biases and distribution shift are often discussed in the context of naturally occurring changes. This is usually referred to as the *average* case. But how bad can this get in the *worst* case?

- Where can changes be adversarial?
- How do we make adversarial changes?
- What is the difference between adversarial and average case?

1 Adversarial changes

Adversarial changes are not necessarily that different from biases and distribution shifts. In fact, adversarial settings can often share the same settings. The key change in the adversarial regime is that instead of average case performance (i.e. over a distribution sh ift or a set of biases), we instead look at worst case performance.

1.1 Per-example changes

Adversarial examples are those that maximize the loss of a model subject to some input constraint.

$$x_{adv} = \max_{\delta \in \mathcal{B}(x)} \ell(f(x), y) \tag{1}$$

The ball of local changes determines the type of adversarial example and is often referred to as the *threat model.* For example

1. Noise model: $\mathcal{B}(x) = \{x + \delta : \|\delta\|_p \leq \epsilon$. Usually invisible noise for $p \in \{1, 2, \infty\}$.
2. Patch model: $\mathcal{B}(x) = \{x + p : p \text{ is a patch}\}$. Unconstrained changes within a region of fixed contiguous size for images
3. Substitutions: $\mathcal{B}(x) = \{y : x \text{ is a synonym of } y\}$. Word substitutions with synonyms for text.

Finding an adversarial example. Let's first consider the noise model in the vision setting. Earliest way to find an adversarial example is FGSM. It is a one-step, first order approximation to the adversarial example optimization problem. For ℓ_∞ noise, this is

$$\delta_{fgsm} = \epsilon \cdot \text{sign}(g) \tag{2}$$

where $g = \nabla_\delta \ell(f(x + \delta), y)$. This is also known as steepest descent, i.e.

$$\epsilon \cdot \text{sign}(g) = \text{argmax}_{\|v\| \leq \epsilon} v^T g \tag{3}$$

Typically we also clip the input to be the range of real pixels, i.e. $[0, 1]$.

Multi-step attack We can run several of these iterations with smaller step sizes. Since these iterates can leave the original ball $\mathcal{B}(x)$, we also project back onto this space (projected steepest descent), i.e.

$$\delta^{t+1} = \delta^t + \text{Proj}_{\mathcal{B}(x)}(\delta^t + \alpha \cdot \text{sign}(\nabla_\delta \ell(f(x^t + \delta), y))) \tag{4}$$

for some step size α. This is usually much more effective in practice. The projection step for the ℓ_∞ ball with radius ϵ is simply to clamp the inputs at $[-\epsilon, \epsilon]$. This can be implemented in a few lines of code:

```
def pgd_linf(model, X, y, epsilon, alpha, num_iter):
    """ Construct FGSM adversarial examples on the examples X"""
    delta = torch.zeros_like(X, requires_grad=True)
    for t in range(num_iter):
        loss = nn.CrossEntropyLoss()(model(X + delta), y)
        loss.backward()
        delta.data = (delta + alpha*delta.grad.detach().sign()).clamp(-epsilon,epsilon)
        delta.grad.zero_()
    return delta.detach()
```

Perceptual threat model

$$\mathcal{B}(x) = \{x' : d(x, x') \leq \epsilon\} \tag{5}$$

where $d(x, x') = \|\phi(x) - \phi(x')\|_2$, ϕ is the activations of a neural network.

Patch attacks Patch attacks have two main components: the patch itself and the location of the patch. Typically the patch itself can be optimized directly similarly like the PGD attack. However, location is more difficulty:

$$\max_{i,j} \ell(f(x + \text{patch}_{ij}, y) \tag{6}$$

Several strategies:

- Random search
- Numerical gradient optimization

1.2 White vs black box attacks

What can you do with query access only?

- Train surrogate model and transfer
- Random search
- Localized random search (square attack)

Character subsitutions This threat model simulates a "typo".

Word substitutions lzantot et al (2018) find word subsitutions by:

- Use the nearest neighbors (i.e. $k = 8$) of a word embedded in a “counter-fitted” word vector spsace, where antonynms are trained to be far apart.
- Neighbor must lie in Euclidean distance threshold
- Word must not decrease the log likelihood of the text under a pretrained language model by too much

2 Distributional changes

The previous section focused on per-example adversarial attacks, i.e. for a given example x find a perturbation x' that is close to x but with different predicted output. Here, the adversary attacks a single example at test time, sometimes referred to as an evasion attack. However, the machine learning pipeline is not just vulnerable at deployment—systems can also be compromised at training time.

How can this happen, i.e. how could an adversary affect your training data?

- Scraping images from the web
- Data harvesting (i.e. spam filters)
- Federated learning

2.1 Data poisoning

In datra poisoining, the attacker is allowed to inject poisoned data into a clean dataset. In this setting:

- Clean image is labeled "correctly"
- Performance changes on a selected target example

Collision attack For a target t and a base image b:

$$p = \arg\min_x \|f(x) - f(t)\|^2 + \beta\|x - b\| \tag{7}$$

This attack assumes you know the feature extractor f. When we don’t know f, we could use a surrogate model, similar to what can be done with adversarial examples. However, data poisoning is much more sensitive to inaccuracies in the surrogate model—small differences can result in a lower rate of poison success. To get around this, one can simply inject multiple poisons over several datasets, in the hopes that the true extracted feature lies within the polytope of the poisoned examples.

This procedure can be generalized to any training algorithm, base images, and attacker objective via meta poisoning. Here, we tie the poison optimization problem into the training loop of SGD:

- Backpropagte the standard loss $\theta = \theta - \alpha \nabla_\theta \ell(\theta, x)$
- Update the poisons with adversarial loss $p = p - \alpha \nabla_p \ell_{adv}(\theta, p)$

2.2 Backdoor attack

In the data poisoning example, we assumed that we, the ones in ch arge of training the model, were the "good guys" and the "bad guys" would try to achieve an adverse outcome by injecting bad data into our training loop. The backdoor setting is similar, but slightly different—in data poisoning, the goal is usually to damage the accuracy of the trained model. On the other hand, in backdoor attacks, the attack does not try to change the predictions on clean examples, but only on backdoored examples. In this sense, a backdoor attack is more invisible and harder to detect (i.e. severe poisoning could be detected if it affects validation accuracy).

To carry out this attack, we simply insert datapoints that have had a trigger added to them with the target class label. Training then proceeds as normal.

In fact, the "bad guys" in backdoor attacks could even control the model and training procedure itself. If someone hands you a "high performing model", you don't know if the model has already been compromised to have adversarial behavior on a specific trigger.

A note on Byzantine attacks. In the Byzantine general problem, in a distributed setting, one assumes that some subset of the works could behave completely arbitrarily. This is related in some sense to the data poisoning or backdoor attacks—both of these are ways to inject arbitrary behaviors via a distributed worker, and both have extensions to the distributed setting.

3 References

Some notes are sourced from Tom Goldstein's tutorial on data poisoning at `https://www.youtube.com/watch?v=MLjK-SC7JSY`.