CHAPTER 12

Adversarial training

As the ultimate goal of defenses is to make models robust against adversarial examples, a very natural idea is to add adversarial examples into the training dataset. This idea, also known as adversarial training, has been implemented in many earlier papers such as (Goodfellow et al., 2015; Kurakin et al., 2017), where they usually generate adversarial examples periodically (e.g., after every few training epochs). However, as the model evolves quickly during the optimization procedure, those adversarial examples usually fail to capture the vulnerable points of the latest model, leading to instabbility and poor performance on robust accuracy (i.e., accuracy against adversarial examples at test time). Later on Madry et al. (2018) showed that adversarial training can be formulated as solving a min-max optimization problem. This simple but insightful formulation enables generating adversarial examples on-the-fly, which significantly improves the robustness of the model. As a result, adversarial training becomes one of the most widely used defense methods.

In this chapter, we will first introduce the min-max formulation of adversarial training in Section 12.1, and then discuss several improvements over vanilla AT in Section 12.2 and 12.3.

12.1 Formulating adversarial training as bilevel optimization

In standard training, the goal is to learn the model parameters θ to minimize the empirical loss, defined as the average loss on all the training examples. The objective of standard training can thus be formulated as solving the following empirical risk minimization problem:

$$\arg\min_{\theta} E_{(x,y)\in\mathcal{D}} L(f_\theta(x), y), \tag{12.1}$$

where E denotes expectation, $(x, y) \in \mathcal{D}$ denotes a pair of dample sample and its label randomly drawn from a distribution $\mathcal{D}$, L is a supervised loss function, and $f_\theta(x)$ is the model's prediction on x.

In adversarial training, we consider that there may be an attacker trying to find the adversarial example with the maximal loss within an ℓ_p ball of

Adversarial Robustness for Machine Learning
https://doi.org/10.1016/B978-0-12-824020-5.00023-5

each example. Therefore, instead of minimizing the regular loss function $L(\cdot, \cdot)$, we want to minimize the loss on the worst-case perturbed example around each natural example. It is thus natural to replace the loss in (12.1) by the maximal loss around each example, which can be written as

$$\arg\min_{\theta} E_{(x,y)\in\mathcal{D}} \max_{x'\in B(x)} L(f_\theta(x'), y), \tag{12.2}$$

where $B(x)$ is the perturbation set around the natural example x. For instance, under a commonly used ℓ_∞ perturbation set, $B(x)$ is the ϵ-bounded ℓ_∞ ball around x, so the objective can be written as

$$\arg\min_{\theta} E_{(x,y)\in\mathcal{D}} \max_{x'\in\|x'-x\|_\infty\leq\epsilon} L(f_\theta(x'), y). \tag{12.3}$$

This formulation is proposed by Madry et al. (2018). This is also called a bilevel optimization as there are two levels (min and max) in the objective. Bilevel optimization problems can often be solved by the alternating minimization procedure: at each update, solve the inner maximization problem to obtain x' and then conduct an outer update on the model parameter θ based on x'.

In alternating minimization, the inner maximization can be viewed as finding an adversarial example of a given input data point x within the perturbation set $B(x)$. In Eq. (12.3), as the inner problem is a constrained maximization task, a natural idea is to solve it by a projected gradient descent (PGD) algorithm, similar to the PGD algorithm when we conduct adversarial attack. In the PGD algorithm, we start from $x_0 = x$ and conduct the following iterative updates:

$$x_t \leftarrow \text{Proj}_{B(x)}(x_{t-1} + \alpha \cdot \text{sign}(\nabla_x L(f_\theta(x_{t-1}), y))), \tag{12.4}$$

where the operator Proj projects the input vector into the constraint set $B(x)$, in this case the ℓ_∞ ball around x. This can be done by clipping each input dimension by

$$\text{Proj}(u)_i = \min(x_i + \epsilon, \max(x_i - \epsilon, u_i))$$

for a given vector u. If we run the inner maximization for k iterations, then we set x' as the output of the kth PGD step and use it to conduct the parameter update of θ. Ideally, a larger k will lead to better results but also consumes more time. If we run PGD for k iterations, then the computational time of the whole adversarial training process will be roughly k times

more than the standard training, which makes adversarial training slow. We will discuss some ways to overcome this problem in the next section.

After obtaining x', we will run one update to solve the outer minimization problem to obtain a better parameter θ. This can be done by computing the gradient of θ with respect to x' ($\nabla_\theta L(f_\theta(x'), y)$) and then pass this gradient to the update rule optimizers (e.g., (SGD or Adam) for the parameter update. The overall PGD-based adversarial training algorithm can be found in Algorithm 3.

Algorithm 3 PGD-based adversarial training (Madry et al., 2018).

Input: Initial model parameter θ_0, number of PGD steps k
for $t = 0, 1, 2, \ldots$ **do**
  Sample (x, y) from training data $\mathcal{D}$
  Let $x_0 = x$
  for $t' = 0, \ldots, k-1$ **do**
    $x_{t'+1} \leftarrow \text{Proj}_{B(x)}(x_{t'} + \alpha \cdot \text{sign}(\nabla_x L(f_\theta(x_{t'}), y)))$
  end for
  Update θ by $\theta \leftarrow \theta - \eta \nabla_\theta L(f_\theta(x_k), y)$
end for

12.2 Faster adversarial training

As mentioned in the previous section, PGD-based adversarial training requires k inner steps, which will increase the training time by k times over standard training. To reduce this overhead, several methods have been developed to achieve a better tradeoff between the robust accuracy and training speed.

Recall that in adversarial training the model update requires gradient of loss function with respect to θ, whereas the PGD updates rely on the gradient of the same loss function with respect to x. These two gradients, denoted as $\nabla_\theta L(f_\theta(x), y)$ and $\nabla_x L(f_\theta(x), y)$, compute the gradient of the same function, but with respect to different variables. In neural networks, gradient is usually computed by back-propagation, and these two gradients can actually share the whole back-propagation process – after computing $\nabla_\theta L(f_\theta(x), y)$, only one additional linear projection step can compute $\nabla_x L(f_\theta(x), y)$. This means that computing both $\nabla_\theta L(f_\theta(x), y)$ and $\nabla_x L(f_\theta(x), y)$ together will have the same cost as computing each of them individually. Motivated by this observation, we can make additional updates during inner PGD steps. In the original adversarial training, each PGD step

computes $\nabla_x L(f_\theta(x), y)$ and only using it to update x. However, as we can also obtain $\nabla_\theta L(f_\theta(x), y)$ with almost zero cost, we can also update θ at each PGD iteration. These simultaneous updates of θ and x will speedup the convergence since we make more updates to θ. At each time, when we find a perturbation (with one PGD step), we immediately update θ to make it performs better on this adversarial example. The resulting algorithm, also known as Free Adversarial Training proposed by Shafahi et al. (2019), thus requires less inner steps to converge to a competitive robust model as standard adversarial training. The algorithm of Free Adversarial Training is described in Algorithm 4.

Algorithm 4 Free adversarial training (Shafahi et al., 2019).

Input: Initial model parameter θ_0, number of PGD steps m
for $t = 0, 1, 2, \ldots$ **do**
  Sample (x, y) from training data $\mathcal{D}$
  Let $x_0 = x$
  for $t' = 0, \ldots, m-1$ **do**
    $x_{t'+1} \leftarrow \text{Proj}_{B(x)}(x_{t'} + \alpha \cdot \text{sign}(\nabla_x L(f_\theta(x_{t'}), y)))$
    Update θ by $\theta \leftarrow \theta - \eta \nabla_\theta L(f_\theta(x_{t'}), y)$
  end for
end for

Several other algorithms also exploited the back-propagation process to speed up adversarial training. See, for instance, (Zhang et al., 2019a). However, more recently, it has been shown that a proper implementation of PGD-based adversarial training with only a single step ($k = 1$) can already achieve good performance. The main trick is that instead of initializing the PGD iterate x_0 from x, we should initialize it from x plus a random noise. Wong et al. (2020a) showed that if we initialize PGD attack from x plus a random vector sampled from Gaussian, adversarial training with a single step can outperform many strong baselines (e.g., FreeAdvTrain). This algorithm with 1-step PGD inner iteration and random initialization is known as Fast Adversarial Training. By further controlling the PGD steps we can then control the tradeoff between speed and robust accuracy.

12.3 Improvements on adversarial training

In theory, robust error can be bounded by two factors: one corresponds to the error on the natural input data, and another corresponds to how

much the prediction can be changed when perturbing an input within an ϵ ball. Inspired by this observation, Zhang et al. (2019b) modify the bilevel objective function of adversarial training as follows:

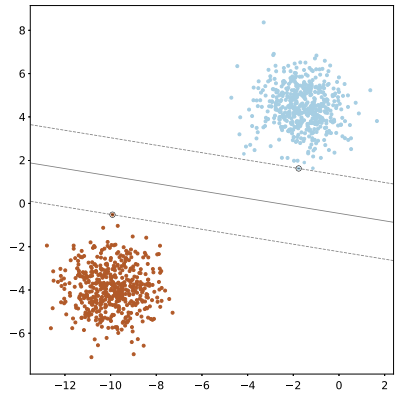
$$\min_{\theta} E_{(x,y)\in\mathcal{D}} L(f_\theta(x), y) + \lambda \max_{x'\in B(x)} L'(f_\theta(x'), f_\theta(x)), \tag{12.5}$$

where L and L' are two loss functions. In this formulation, we assume both loss functions are cross-entropy loss, so the first term is the standard training loss defined on the clean data, and the second term measures the difference between the prediction of clean example ($f(x)$) and the perturbed example ($f(x')$); λ is a positive constant that controls the balance between two terms. By decomposing the clean and robust loss it becomes easier to control the robustness and accuracy tradeoff (with a properly chosen λ). This formulation, also known as *TRADES*, is shown to achieve better performance than adversarial training and was the winner of the NeurIPS 2018 defense challenge.

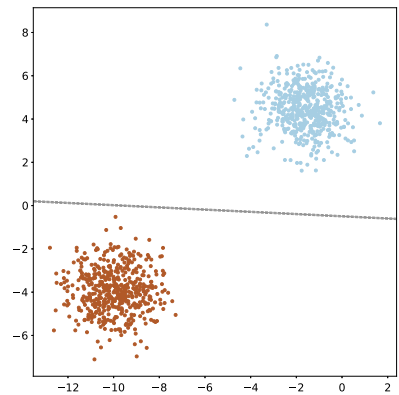
Later on, several improvements have been made over TRADES. In particular, many papers found that one has to weight each sample differently to obtain the best performance. There are several motivations behind this. First, we should treat misclassified samples differently from correctly classified ones. For misclassified samples, as the classifier already predicts the wrong label, it is not useful to make the model robust on those points. Second, some examples are intrinsically hard to classify (e.g., examples that are close to decision boundary), so forcing them to be robust will result in poor performance. For instance, in Fig. 12.1, we showed a linear classifier, where the true margin between two classes are 2.5, and there does not exist a classifier that can robustly classify all the samples with $\epsilon = 3$. If we train the classifier with $\epsilon = 3$, then the result will be a poor classifier as shown in Fig. 12.1(c).

As a result, many formulations have been developed to conduct adversarial training with nonuniform weights among samples. For instance, Wang et al. (2019d) proposed a weighted version of TRADES, where they weight the samples by the correctness of each prediction. More specifically, let $p_y(x, \theta) := (f_\theta(x'))_y$ be the normalized prediction probability for label y – it is the softmax output of the model between $[0, 1]$. Then we can consider a weighted version of the TRADES objective as

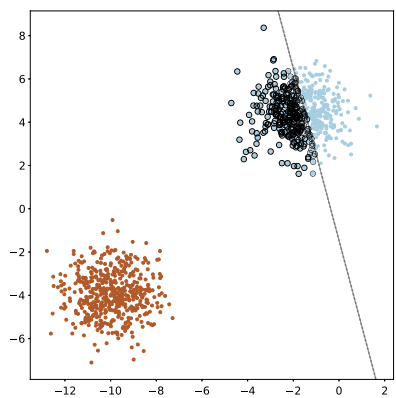
$$\min_{\theta} E_{(x,y)\in\mathcal{D}} L(f_\theta(x), y) + \lambda \max_{x'\in B(x)} L(f_\theta(x'), f_\theta(x))(1 - p_y(x, \theta)), \tag{12.6}$$



(a) Standard training



(b) Adv-train with $\epsilon = 1$



(c) Adv-train with $\epsilon = 4$

Figure 12.1 Different training methods on a linearly separable binary classification dataset with 1.75 margin for both classes. Adversarial training with small ϵ works fine, but for a large ϵ beyond the true margin, adversarial training would ruin the classifier's classification performance.

where the second term is weighted by $1 - p_y(x, \theta)$, which is close to 0 if the sample is correctly classified and to 1 if it is (more) misclassified. This encourages the network to make more progress to the misclassified data points.

Based on a similar idea, we can also have adaptive ϵ for each sample. A sample that is misclassified should have smaller ϵ since it is not important (or not possible) to make it robust, whereas a corrected classified should be assigned with larger ϵ. Several methods have been proposed in this direction, including (Ding et al., 2018; Balaji et al., 2019; Cheng et al., 2020b).

12.4 Extended reading

- Wang et al. (2019d) developed algorithmic convergence analysis for adversarial training.
- Zhang et al. (2022) proposed distributed training algorithms for scaling up adversarial training.
- Cheng et al. (2021) proposed a self-progressing robust training method.